
invenio Documentation

Release 3.0.0a5.dev20161219

CERN

Mar 24, 2017

Contents

1	Overview	3
1.1	What is Invenio?	3
1.2	Who is using Invenio?	4
1.3	Digital repositories	4
1.4	Use cases	5
1.5	History	5
2	User's Guide	7
2.1	Quickstart	7
2.2	First steps	8
2.3	Running Invenio	29
2.4	Loading content	32
2.5	Accessing content	34
2.6	Enabling ORCID login	37
2.7	Migrating to v3	40
3	Deployment Guide	43
3.1	Overview	43
3.2	Services	43
3.3	Detailed installation guide	50
3.4	Configuration	65
3.5	Daemonizing applications	65
3.6	Monitoring	65
3.7	High Availability	65
3.8	Deploying with Fabric	65
3.9	Deploying with Docker	65
4	Developer's Guide	67
4.1	First steps	67
4.2	Setting up your development environment	78
4.3	System architecture	78
4.4	Application architecture	79
4.5	Extending Invenio	83
4.6	Invenio module layout	83
4.7	Bundles	86
4.8	Creating a data model	86
4.9	Docker	93

4.10	W6. Rebasing against latest git/master	97
4.11	Learning Python	98
5	Community	99
5.1	Getting help	99
5.2	Communication channels	100
5.3	Contribution guide	101
5.4	Style guide TODO	104
5.5	Translation guide	105
5.6	Maintainer's guide TODO	106
5.7	Releases	111
5.8	Code of Conduct	114
5.9	License	115
5.10	Authors	115
5.11	TODO	121

Invenio is a digital library framework that enables you to build and run your own large scale digital repository.

Invenio can be used either as an underlying framework or via the currently one pre-packaged flavour which is ready to run out-of-the-box:

- Invenio v3 Integrated Library System (ILS)

Three further pre-packaged flavours are planned:

- Invenio v3 Research Data Management (RDM)
- Invenio v3 Media Archive (MA)
- Invenio v3 Institutional Repository (IR)

Invenio is currently powering some of the largest digital repositories in the world such as:

- [CERN Document Server \(v1\)](#)
- [CERN Open Data Repository \(v2\)](#)
- [INSPIRE High-Energy Physics Repository \(v1\)](#)
- [Caltech Library Catalog \(v1\)](#)
- [DESY Publication Database \(v1\)](#)
- [Zenodo research data repository \(v3\)](#)

Warning: Invenio v3 is still under development. We expect the first stable release by end-April 2017.
--

What is Invenio?

Todo

Fix up this section

Features

Navigable collection tree

- Documents organised in collections
- Regular and virtual collection trees
- Customizable portalboxes for each collection
- At CERN, over 1,000,000 documents in 700 collections

Powerful search engine

- Specially designed indexes to provide fast search speed for repositories of up to 2,000,000 records
- Customizable simple and advanced search interfaces
- Combined metadata, fulltext and citation search in one go
- Results clustering by collection

Flexible metadata

- Standard metadata format (MARC)
- Handling articles, books, theses, photos, videos, museum objects and more
- Customizable display and linking rules

User personalization

- user-defined document baskets
- user-defined automated email notification alerts
- basket-sharing within user groups
- Amazon-like user comments for documents in repository and shared baskets

Multiple output formats



Who is using Invenio?

Todo

Write this section

Digital repositories

Todo

Write this section

What is a digital repository?

Record keeping

Persistent identifiers

Metadata

Data

Open Archival Information Systems (OAIS)

Use cases

Todo

Write this section

Integrated library system (ILS)

Research data management (RDM)

Institutional repository (IR)

Multimedia archive

History

Invenio v3 for v1 users

Invenio v3 is a completely new application that has been rewritten from scratch in roughly a year. Why such a dramatic decision? To understand why the rewrite was necessary we have to go back to when Invenio was called CDSWare, back to August 1st 2002 when the first version of Invenio was released.

In 2002:

- First iPod had just hit the market (Nov 2001).
- The Budapest Open Access Initiative had just been signed (Feb, 2002).
- JSON had just been discovered (2001).
- Python 2.1 had just been released (2001).
- Apache Lucene had just joined the Apache Jakarta project (but not yet an official top-level project).
- MySQL v4.0 beta was released and did not even have transactions yet.
- Hibernate ORM was released.
- The first DOI had just been assigned to a dataset.

Following products did not even exist:

- Apache Solr (2004)
- Google Maps (2005)
- Google Scholar (2004)
- Facebook (2007)
- Django (2005)

A lot has happen in the past 15 years. Many problems that Invenio originally had to deal with now have open source off-the-shelf solutions available. In particular two things happen:

- Search become pervasive with the exponential growth of data collected and created on the internet every day, and open source products to solve handles these needs like Elasticsearch became big business.
- Web frameworks for both front-end and back-end made it significant faster to develop web applications.

What happened to Invenio v2?

Initial in 2011 we started out on creating a hybrid application which would allow us to progressively migrate features as we had the time. In 2013 we launched Zenodo as the first site on the v2 development version which among other things featured Jinja templates instead of the previous Python based templates.

In theory everything was sound, however over the following years it became very difficult to manage the inflow of changes from larger and larger teams on the development side and operationally proved to be quite unstable compared to v1.

Last but not least, Invenio v1 was built in a time where the primary need was publication repositories and v2 inherited this legacy making it difficult to deal with very large research datasets.

Thus, in late 2015 we were being slowed so much down by our past legacy that we saw no other way that starting over from scratch if we were to deal with the next 20 years of challenges.

This part of the documentation will show you how to get started using Invenio.

Quickstart

Using Docker

You can get Invenio v3.0 demo site up and running using Docker:

```
docker-compose build
docker-compose up -d
docker-compose run --rm web ./scripts/populate-instance.sh
firefox http://127.0.0.1/records/1
```

Using Vagrant

You can get Invenio v3.0 demo site up and running using Vagrant:

```
vagrant up
vagrant ssh web -c 'source .inveniorc && /vagrant/scripts/create-instance.sh'
vagrant ssh web -c 'source .inveniorc && /vagrant/scripts/populate-instance.sh'
vagrant ssh web -c 'source .inveniorc && nohup /vagrant/scripts/start-instance.sh'
firefox http://192.168.50.10/records/1
```

Using kickstart scripts

You can set some environment variables and run kickstart provisioning and installation scripts manually:

```
vim .inveniorc
source .inveniorc
scripts/provision-web.sh
scripts/provision-postgresql.sh
scripts/provision-elasticsearch.sh
scripts/provision-redis.sh
scripts/provision-rabbitmq.sh
scripts/provision-worker.sh
scripts/create-instance.sh
scripts/populate-instance.sh
scripts/start-instance.sh
firefox http://192.168.50.10/records/1
```

See *Install prerequisites* for more information.

First steps

Install prerequisites

Overview

Invenio uses several services such as [PostgreSQL](#) database server, [Redis](#) for caching, [Elasticsearch](#) for indexing and information retrieval, [RabbitMQ](#) for messaging, [Celery](#) for task execution.

You can install them manually or you can use provided kickstart scripts as mentioned in *Quickstart*:

```
vim .inveniorc
source .inveniorc
scripts/provision-web.sh
scripts/provision-postgresql.sh
scripts/provision-elasticsearch.sh
scripts/provision-redis.sh
scripts/provision-rabbitmq.sh
scripts/provision-worker.sh
```

The next section gives a detailed walk-through of what the scripts do.

Concrete example

In this installation example, we'll create an Invenio digital library instance using a multi-machine setup where separate services (such as the database server and the web server) run on separate dedicated machines. Such a multi-machine setup emulates to what one would typically use in production. (However, it is very well possible to follow this guide and install all the services onto the same “localhost”, if one wants to.)

We'll use six dedicated machines running the following services:

node	IP	runs
web	192.168.50.10	Invenio web application
postgresql	192.168.50.11	PostgreSQL database server
redis	192.168.50.12	Redis caching service
elasticsearch	192.168.50.13	Elasticsearch information retrieval service
rabbitmq	192.168.50.14	RabbitMQ messaging service
worker	192.168.50.15	Celery worker node

The instructions below are tested on Ubuntu 14.04 LTS (Trusty Tahr) and CentOS 7 operating systems. For other operating systems such as Mac OS X, you may want to check out the “kickstart” set of scripts coming with the Invenio source code that perform the below-quoted installation steps in an unattended automated way.

Environment variables

Let’s define some useful environment variables that will describe our Invenio instance setup:

INVENIO_WEB_HOST The IP address of the Web server node.

INVENIO_WEB_INSTANCE The name of your Invenio instance that will be created. Usually equal to the name of the Python virtual environment.

INVENIO_WEB_VENV The name of the Python virtual environment where Invenio will be installed. Usually equal to the name of the Invenio instance.

INVENIO_USER_EMAIL The email address of a user account that will be created on the Invenio instance.

INVENIO_USER_PASS The password of this Invenio user.

INVENIO_POSTGRESQL_HOST The IP address of the PostgreSQL database server.

INVENIO_POSTGRESQL_DBNAME The database name that will hold persistent data of our Invenio instance.

INVENIO_POSTGRESQL_DBUSER The database user name used to connect to the database server.

INVENIO_POSTGRESQL_DBPASS The password of this database user.

INVENIO_REDIS_HOST The IP address of the Redis server.

INVENIO_ELASTICSEARCH_HOST The IP address of the Elasticsearch information retrieval server.

INVENIO_RABBITMQ_HOST The IP address of the RabbitMQ messaging server.

INVENIO_WORKER_HOST The IP address of the Celery worker node.

In our example setup, we shall use:

```
export INVENIO_WEB_HOST=192.168.50.10
export INVENIO_WEB_INSTANCE=invenio
export INVENIO_WEB_VENV=invenio
export INVENIO_USER_EMAIL=info@inveniosoftware.org
export INVENIO_USER_PASS=uspass123
export INVENIO_POSTGRESQL_HOST=192.168.50.11
export INVENIO_POSTGRESQL_DBNAME=invenio
export INVENIO_POSTGRESQL_DBUSER=invenio
export INVENIO_POSTGRESQL_DBPASS=dbpass123
export INVENIO_REDIS_HOST=192.168.50.12
export INVENIO_ELASTICSEARCH_HOST=192.168.50.13
export INVENIO_RABBITMQ_HOST=192.168.50.14
export INVENIO_WORKER_HOST=192.168.50.15
```

Let us save this configuration in a file called `.inveniorc` for future use.

Web

The web application node (192.168.50.10) is where the main Invenio application will be running. We need to provision it with some system dependencies in order to be able to install various underlying Python and JavaScript libraries.

The web application node can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-web.sh
```

Let's see in detail what the web provisioning script does.

First, let's see if using `sudo` will be required:

```
# runs as root or needs sudo?
if [[ "$EUID" -ne 0 ]]; then
    sudo='sudo'
else
    sudo=''
fi
```

Second, some useful system tools are installed:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
# update list of available packages:
$sudo apt-get -y update

# install useful system tools:
$sudo apt-get -y install \
    curl \
    git \
    rlwrap \
    screen \
    vim
```

- on CentOS 7:

```
# add EPEL external repository:
$sudo yum install -y epel-release

# install useful system tools:
$sudo yum install -y \
    curl \
    git \
    rlwrap \
    screen \
    vim
```

Third, an external Node.js package repository is enabled. We'll be needing to install and run Npm on the web node later. The Node.js repository is enabled as follows:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
if [[ ! -f /etc/apt/sources.list.d/nodesource.list ]]; then
    curl -sL https://deb.nodesource.com/setup_4.x | $sudo bash -
fi
```

- on CentOS 7:

```
if [[ ! -f /etc/yum.repos.d/nodesource-el.repo ]]; then
  curl -sL https://rpm.nodesource.com/setup_4.x | $sudo bash -
fi
```

Fourth, all the common prerequisite software libraries and packages that Invenio needs are installed:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
$sudo apt-get -y install \
  libffi-dev \
  libfreetype6-dev \
  libjpeg-dev \
  libmsgpack-dev \
  libssl-dev \
  libtiff-dev \
  libxml2-dev \
  libxslt-dev \
  nodejs \
  python-dev \
  python-pip
```

- on CentOS7:

```
# install development tools:
$sudo yum update -y
$sudo yum groupinstall -y "Development Tools"
$sudo yum install -y \
  libffi-devel \
  libxml2-devel \
  libxslt-devel \
  openssl-devel \
  policycoreutils-python \
  python-devel \
  python-pip
$sudo yum install -y --disablerepo=epel \
  nodejs
```

We want to use PostgreSQL database in this installation example, so we need to install corresponding libraries too:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
$sudo apt-get -y install \
  libpq-dev
```

- on CentOS7:

```
$sudo yum install -y \
  postgresql-devel
```

Fifth, now that Node.js is installed, we can proceed with installing Npm and associated CSS/JS filter tools. Let's do it globally:

- on either of the operating systems:

```
# $sudo su -c "npm install -g npm"
$sudo su -c "npm install -g node-sass@3.8.0 clean-css@3.4.12 requirejs uglify-js"
```

Sixth, we'll install Python virtual environment wrapper tools and activate them in the current user shell process:

- on either of the operating systems:

```
$sudo pip install -U setuptools pip
$sudo pip install -U virtualenvwrapper
if ! grep -q virtualenvwrapper ~/.bashrc; then
    mkdir -p "$HOME/.virtualenvs"
    echo "export WORKON_HOME=$HOME/.virtualenvs" >> "$HOME/.bashrc"
    echo "source $(which virtualenvwrapper.sh)" >> "$HOME/.bashrc"
fi
export WORKON_HOME=$HOME/.virtualenvs
# shellcheck source=/dev/null
source "$(which virtualenvwrapper.sh)"
```

Seventh, we install Nginx web server and configure appropriate virtual host:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
# install Nginx web server:
$sudo apt-get install -y nginx

# configure Nginx web server:
$sudo cp -f "$scriptpathname/./nginx/invenio.conf" /etc/nginx/sites-available/
$sudo sed -i "s,/home/invenio/,/home/${whoami}/,g" /etc/nginx/sites-available/
↪invenio.conf
$sudo rm /etc/nginx/sites-enabled/default
$sudo ln -s /etc/nginx/sites-available/invenio.conf /etc/nginx/sites-enabled/
$sudo /usr/sbin/service nginx restart
```

- on CentOS7:

```
# install Nginx web server:
$sudo yum install -y nginx

# configure Nginx web server:
$sudo cp "$scriptpathname/./nginx/invenio.conf" /etc/nginx/conf.d/
$sudo sed -i "s,/home/invenio/,/home/${whoami}/,g" /etc/nginx/conf.d/invenio.conf

# add SELinux permissions if necessary:
if $sudo getenforce | grep -q 'Enforcing'; then
    if ! $sudo semanage port -l | tail -n +1 | grep -q '8888'; then
        $sudo semanage port -a -t http_port_t -p tcp 8888
    fi
    if ! $sudo semanage port -l | grep ^http_port_t | grep -q 5000; then
```



```

        $sudo semanage port -m -t http_port_t -p tcp 5000
    fi
    if ! $sudo getsebool -a | grep httpd | grep httpd_enable_homedirs | grep -q_
↪on; then
        $sudo setsebool -P httpd_enable_homedirs 1
        mkdir -p "/home/$(whoami)/.virtualenvs/${INVENIO_WEB_VENV}/var/instance/
↪static"
        $sudo chcon -R -t httpd_sys_content_t "/home/$(whoami)/.virtualenvs/$
↪{INVENIO_WEB_VENV}/var/instance/static"
    fi
fi

$sudo sed -i 's,80,8888,g' /etc/nginx/nginx.conf
$sudo chmod go+rx "/home/$(whoami)/"
$sudo /sbin/service nginx restart

# open firewall ports:
if firewall-cmd --state | grep -q running; then
    $sudo firewall-cmd --permanent --zone=public --add-service=http
    $sudo firewall-cmd --permanent --zone=public --add-service=https
    $sudo firewall-cmd --reload
fi

```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
$sudo apt-get -y autoremove && $sudo apt-get -y clean
```

- on CentOS7:

```
$sudo yum clean -y all
```

Database

The database server (192.168.50.11) will hold persistent data of our Invenio installation, such as bibliographic records or user accounts. Invenio supports MySQL, PostgreSQL, and SQLite databases. In this tutorial, we shall use PostgreSQL that is the recommended database platform for Invenio.

The database server node can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-postgresql.sh
```

Let's see in detail what the database provisioning script does.

First, we install and configure the database software:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
# update list of available packages:
```

```

sudo DEBIAN_FRONTEND=noninteractive apt-get -y update

# install PostgreSQL:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y install \
    postgresql

# allow network connections:
if ! grep -q "listen_addresses.*${INVENIO_POSTGRESQL_HOST}" \
    /etc/postgresql/9.3/main/postgresql.conf; then
    echo "listen_addresses = '${INVENIO_POSTGRESQL_HOST}'" | \
        sudo tee -a /etc/postgresql/9.3/main/postgresql.conf
fi

# grant access rights:
if ! sudo grep -q "host.*${INVENIO_POSTGRESQL_DBNAME}.*${INVENIO_POSTGRESQL_
↪DBUSER}" \
    /etc/postgresql/9.3/main/pg_hba.conf; then
    echo "host ${INVENIO_POSTGRESQL_DBNAME} ${INVENIO_POSTGRESQL_DBUSER} $
↪${INVENIO_WEB_HOST}/32 md5" | \
        sudo tee -a /etc/postgresql/9.3/main/pg_hba.conf
fi

# grant database creation rights via SQLAlchemy-Utills:
if ! sudo grep -q "host.*templatel.*${INVENIO_POSTGRESQL_DBUSER}" \
    /etc/postgresql/9.3/main/pg_hba.conf; then
    echo "host templatel ${INVENIO_POSTGRESQL_DBUSER} ${INVENIO_WEB_HOST}/32 md5"
↪ | \
        sudo tee -a /etc/postgresql/9.3/main/pg_hba.conf
fi

# restart PostgreSQL server:
sudo /etc/init.d/postgresql restart

```

- on CentOS 7:

```

# add EPEL external repository:
sudo yum install -y epel-release

# install PostgreSQL:
sudo yum update -y
sudo yum install -y \
    postgresql-server

# initialise PostgreSQL database:
sudo -i -u postgres pg_ctl initdb

# allow network connections:
if ! sudo grep -q "listen_addresses.*${INVENIO_POSTGRESQL_HOST}" \
    /var/lib/pgsql/data/postgresql.conf; then
    echo "listen_addresses = '${INVENIO_POSTGRESQL_HOST}'" | \
        sudo tee -a /var/lib/pgsql/data/postgresql.conf
fi

# grant access rights:
if ! sudo grep -q "host.*${INVENIO_POSTGRESQL_DBNAME}.*${INVENIO_POSTGRESQL_
↪DBUSER}" \

```

```

    /var/lib/pgsql/data/pg_hba.conf; then
    echo "host ${INVENIO_POSTGRESQL_DBNAME} ${INVENIO_POSTGRESQL_DBUSER} $
↪ ${INVENIO_WEB_HOST}/32 md5" | \
        sudo tee -a /var/lib/pgsql/data/pg_hba.conf
    fi

    # grant database creation rights via SQLAlchemy-Utils:
    if ! sudo grep -q "host.*templatel.*${INVENIO_POSTGRESQL_DBUSER}" \
        /var/lib/pgsql/data/pg_hba.conf; then
    echo "host templatel ${INVENIO_POSTGRESQL_DBUSER} ${INVENIO_WEB_HOST}/32 md5"
↪ | \
        sudo tee -a /var/lib/pgsql/data/pg_hba.conf
    fi

    # open firewall ports:
    if firewall-cmd --state | grep -q running; then
        sudo firewall-cmd --zone=public --add-service=postgresql --permanent
        sudo firewall-cmd --reload
    fi

    # enable PostgreSQL upon reboot:
    sudo systemctl enable postgresql

    # restart PostgreSQL server:
    sudo systemctl start postgresql

```

We can now create a new database user with the necessary access permissions on the new database:

- on either of the operating systems:

```

# create user if it does not exist:
echo "SELECT 1 FROM pg_roles WHERE rolname='${INVENIO_POSTGRESQL_DBUSER}'" | \
    sudo -i -u postgres psql -tA | grep -q 1 || \
    echo "CREATE USER ${INVENIO_POSTGRESQL_DBUSER} WITH PASSWORD '${INVENIO_
↪ POSTGRESQL_DBPASS}';" | \
        sudo -i -u postgres psql

# create database if it does not exist:
echo "SELECT 1 FROM pg_database WHERE datname='${INVENIO_POSTGRESQL_DBNAME}'" | \
    sudo -i -u postgres psql -tA | grep -q 1 || \
    echo "CREATE DATABASE ${INVENIO_POSTGRESQL_DBNAME};" | \
        sudo -i -u postgres psql

# grant privileges to the user on this database:
echo "GRANT ALL PRIVILEGES ON DATABASE ${INVENIO_POSTGRESQL_DBNAME} TO ${INVENIO_
↪ POSTGRESQL_DBUSER};" | \
        sudo -i -u postgres psql

```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
sudo apt-get -y autoremove && sudo apt-get -y clean
```

- on CentOS7:

```
sudo yum clean -y all
```

Redis

The Redis server (192.168.50.12) is used for various caching needs.

The Redis server can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-redis.sh
```

Let's see in detail what the Redis provisioning script does.

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
# update list of available packages:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y update

# install Redis server:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y install \
    redis-server

# allow network connections:
if ! grep -q "${INVENIO_REDIS_HOST}" /etc/redis/redis.conf; then
    sudo sed -i "s/bind 127.0.0.1/bind 127.0.0.1 ${INVENIO_REDIS_HOST}/" \
        /etc/redis/redis.conf
fi

# restart Redis server:
sudo /etc/init.d/redis-server restart
```

- on CentOS 7:

```
# add EPEL external repository:
sudo yum install -y epel-release

# update list of available packages:
sudo yum update -y

# install Redis server:
sudo yum install -y \
    redis

# allow network connections:
if ! grep -q "${INVENIO_REDIS_HOST}" /etc/redis.conf; then
    sudo sed -i "s/bind 127.0.0.1/bind 127.0.0.1 ${INVENIO_REDIS_HOST}/" \
        /etc/redis.conf
fi

# open firewall ports:
```

```

if firewall-cmd --state | grep -q running; then
    sudo firewall-cmd --zone=public --add-port=6379/tcp --permanent
    sudo firewall-cmd --reload
fi

# enable Redis upon reboot:
sudo systemctl enable redis

# start Redis:
sudo systemctl start redis

```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
sudo apt-get -y autoremove && sudo apt-get -y clean
```

- on CentOS7:

```
sudo yum clean -y all
```

Elasticsearch

The Elasticsearch server (192.168.50.13) is used to index and search bibliographic records, fulltext documents, and other various interesting information managed by our Invenio digital library instance.

The Elasticsearch server can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-elasticsearch.sh
```

Let's see in detail what the Elasticsearch provisioning script does.

- on Ubuntu 14.04 LTS (Trusty Tahr):

```

# install curl:
sudo apt-get -y install curl

# add external Elasticsearch repository:
if [[ ! -f /etc/apt/sources.list.d/elasticsearch-2.x.list ]]; then
    curl -sL https://packages.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add
    ↪-
    echo "deb http://packages.elastic.co/elasticsearch/2.x/debian stable main" | \
        sudo tee -a /etc/apt/sources.list.d/elasticsearch-2.x.list
fi

# update list of available packages:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y update

# install Elasticsearch server:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y install \

```

```
    elasticsearch \
    openjdk-7-jre

# allow network connections:
if ! sudo grep -q "network.host: ${INVENIO_ELASTICSEARCH_HOST}" \
    /etc/elasticsearch/elasticsearch.yml; then
    echo "network.host: ${INVENIO_ELASTICSEARCH_HOST}" | \
        sudo tee -a /etc/elasticsearch/elasticsearch.yml
fi

# enable Elasticsearch upon reboot:
sudo update-rc.d elasticsearch defaults 95 10

# start Elasticsearch:
sudo /etc/init.d/elasticsearch restart
```

- on CentOS 7:

```
# add external Elasticsearch repository:
if [[ ! -f /etc/yum.repos.d/elasticsearch.repo ]]; then
    sudo rpm --import \
        https://packages.elastic.co/GPG-KEY-elasticsearch
    echo "[elasticsearch-2.x]
name=Elasticsearch repository for 2.x packages
baseurl=http://packages.elastic.co/elasticsearch/2.x/centos
gpgcheck=1
gpgkey=http://packages.elastic.co/GPG-KEY-elasticsearch
enabled=1" | \
        sudo tee -a /etc/yum.repos.d/elasticsearch.repo
fi

# update list of available packages:
sudo yum update -y

# install Elasticsearch:
sudo yum install -y \
    elasticsearch \
    java

# allow network connections:
if ! sudo grep -q "network.host: ${INVENIO_ELASTICSEARCH_HOST}" \
    /etc/elasticsearch/elasticsearch.yml; then
    echo "network.host: ${INVENIO_ELASTICSEARCH_HOST}" | \
        sudo tee -a /etc/elasticsearch/elasticsearch.yml
fi

# open firewall ports:
if firewall-cmd --state | grep -q running; then
    sudo firewall-cmd --zone=public --add-port=9200/tcp --permanent
    sudo firewall-cmd --reload
fi

# enable Elasticsearch upon reboot:
sudo systemctl enable elasticsearch

# start Elasticsearch:
```

```
sudo systemctl start elasticsearch
```

Some packages require extra plugins to be installed.

```
$sudo /usr/share/elasticsearch/bin/plugin install -b mapper-attachments
```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
sudo apt-get -y autoremove && sudo apt-get -y clean
```

- on CentOS7:

```
sudo yum clean -y all
```

RabbitMQ

The RabbitMQ server (192.168.50.14) is used as a messaging middleware broker.

The RabbitMQ server can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-rabbitmq.sh
```

Let's see in detail what the RabbitMQ provisioning script does.

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
# update list of available packages:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y update

# install RabbitMQ server:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y install \
    rabbitmq-server
```

- on CentOS 7:

```
# add EPEL external repository:
sudo yum install -y epel-release

# update list of available packages:
sudo yum update -y

# install Rabbitmq:
sudo yum install -y \
    rabbitmq-server
```

```
# open firewall ports:
if firewall-cmd --state | grep -q running; then
    sudo firewall-cmd --zone=public --add-port=5672/tcp --permanent
    sudo firewall-cmd --reload
fi

# enable RabbitMQ upon reboot:
sudo systemctl enable rabbitmq-server

# start RabbitMQ:
sudo systemctl start rabbitmq-server
```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
sudo apt-get -y autoremove && sudo apt-get -y clean
```

- on CentOS7:

```
sudo yum clean -y all
```

Worker

The Celery worker node (192.168.50.15) is used to execute potentially long tasks in asynchronous manner.

The worker node can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-worker.sh
```

Let's see in detail what the worker provisioning script does.

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
echo "FIXME worker is a copy of web node"
```

- on CentOS 7:

```
echo "FIXME worker is a copy of web node"
```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
sudo apt-get -y autoremove && sudo apt-get -y clean
```


- on CentOS7:

```
sudo yum clean -y all
```

Install Invenio

Now that all the prerequisites have been set up in *Install prerequisites*, we can proceed with the installation of the Invenio itself. The installation is happening on the web node (192.168.50.10).

We start by creating and configuring a new Invenio instance, continue by populating it with some example records, and finally we start the web application. This can be done in an automated unattended way by running the following scripts:

```
source .inveniorc
./scripts/create-instance.sh
./scripts/populate-instance.sh
./scripts/start-instance.sh
```

Note: If you want to install the very-bleeding-edge Invenio packages from GitHub, you can run the `create-instance.sh` script with the `--devel` argument:

```
./scripts/create-instance.sh --devel
```

Let's see in detail about every Invenio installation step.

Create instance

We start by creating a fresh new Python virtual environment that will hold our brand new Invenio v3.0 instance:

```
mkvirtualenv "${INVENIO_WEB_VENV}"
cdvirtualenv
```

We continue by installing Invenio v3.0 Integrated Library System flavour demo site from PyPI:

```
pip install invenio-app-ils[postgresql,elasticsearch2]
```

Let's briefly customise our instance with respect to the location of the database server, the Redis server, the Elasticsearch server, and all the other dependent services in our multi-server environment:

```
mkdir -p "var/instance/"
pip install "jinja2-cli>=0.6.0"
jinja2 "${scriptpathname}/instance.cfg" > "var/instance/${INVENIO_WEB_INSTANCE}.cfg"
```

In the instance folder, we run Npm to install any JavaScript libraries that Invenio needs:

```
/${INVENIO_WEB_INSTANCE} npm
cdvirtualenv "var/instance/static"
CI=true npm install
```

We can now collect and build CSS/JS assets of our Invenio instance:

```
/${INVENIO_WEB_INSTANCE} collect -v
/${INVENIO_WEB_INSTANCE} assets build
```

Our first new Invenio instance is created and ready for loading some example records.

Populate instance

We proceed by creating a dedicated database that will hold persistent data of our installation, such as bibliographic records or user accounts. The database tables can be created as follows:

```
/${INVENIO_WEB_INSTANCE} db init
/${INVENIO_WEB_INSTANCE} db create
```

We continue by creating a user account:

```
/${INVENIO_WEB_INSTANCE} users create \
    "${INVENIO_USER_EMAIL}" \
    --password "${INVENIO_USER_PASS}" \
    --active
```

We can now create the Elasticsearch indexes and initialise the indexing queue:

```
/${INVENIO_WEB_INSTANCE} index init
sleep 20
/${INVENIO_WEB_INSTANCE} index queue init
```

We proceed by populating our Invenio demo instance with some example demo MARCXML records:

```
/${INVENIO_WEB_INSTANCE} demo init
```

Start instance

Let's now start the web application:

```
/${INVENIO_WEB_INSTANCE} run -h 0.0.0.0 &
```

and the web server:

```
$sudo service nginx restart
```

We should now see our demo records on the web:

```
firefox http://${INVENIO_WEB_HOST}/records/1
```

and we can access them via REST API:

```
curl -i -H "Accept: application/json" \
  http://${INVENIO_WEB_HOST}/api/records/1
```

We are done! Our first Invenio v3.0 demo instance is fully up and running.

Configure Invenio

After having installed Invenio demo site in *Install Invenio*, let us see briefly the instance configuration.

Configure instance

Invenio instance can be configured by editing `invenio.cfg` configuration file. Here is an example:

```
$ cdvirtualenv var/instance/
$ cat invenio.cfg
# Database
SQLALCHEMY_DATABASE_URI='postgresql+psycopy2://invenio:dbpass123@192.168.50.10:5432/
↳invenio'

# Statis files
COLLECT_STORAGE='flask_collect.storage.file'

# Redis
CACHE_TYPE='redis'
CACHE_REDIS_HOST='192.168.50.10'
CACHE_REDIS_URL='redis://192.168.50.10:6379/0'
ACCOUNTS_SESSION_REDIS_URL='redis://192.168.50.10:6379/1'

# Celery
BROKER_URL='amqp://guest:guest@192.168.50.10:5672//'
CELERY_RESULT_BACKEND='redis://192.168.50.10:6379/2'

# Elasticsearch
SEARCH_ELASTIC_HOSTS='192.168.50.10'

# JSON Schema
JSONSCHEMAS_ENDPOINT='/schema'
JSONSCHEMAS_HOST='192.168.50.10'

# OAI server
OAISERVER_RECORD_INDEX='marc21'
OAISERVER_ID_PREFIX='oai:invenio:recid/'
```

Configuration options

Since Invenio demo site uses ILS flavour of Invenio, you can learn more about the configuration options in [Invenio-App-ILS configuration documentation](#).

We shall see how to customise instance more deeply in *Customise Invenio*.

Create and search your first record

Now that Invenio demo site has been installed in *Install Invenio*, let us see how we can load some records.

Upload a new record

For the ILS flavour let us create a small example record in MARCXML format:

```
$ vim book.xml
$ cat book.xml
<?xml version="1.0" encoding="UTF-8"?>
<collection xmlns="http://www.loc.gov/MARC21/slim">
<record>
  <datafield tag="245" ind1=" " ind2=" ">
    <subfield code="a">This is title</subfield>
  </datafield>
  <datafield tag="100" ind1=" " ind2=" ">
    <subfield code="a">Doe, John</subfield>
  </datafield>
</record>
</collection>
```

We can upload it by using `invenio marc21 import` command:

```
$ invenio marc21 import --bibliographic book.xml
Importing records
Created record 117
Indexing records
```

Search for the record

Let us verify that it was well uploaded:

```
$ firefox http://192.168.50.10/records/117
```

and that it is well searchable:

```
$ firefox http://192.168.50.10/search?q=doe
```

Uploading content

For more information on how to upload content to an Invenio instance, see the documentation chapter on *Loading content*.

Customise Invenio

The goal of this tutorial is to demonstrate basic Invenio customisation. We shall modify the size logo, the page templates, the search facets, the sort options, and more.

Install “beauty” module

First go in the virtual machine:

```
laptop> vagrant ssh web
vagrant> workon invenio
```

Install the module invenio-beauty:

```
vagrant> cd /vagrant/2-customization/invenio-beauty
vagrant> pip install .
vagrant> invenio collect
vagrant> invenio run -h 0.0.0.0
```

Customize logo and templates

If you go to <http://192.168.50.10/>, you will see the default Invenio, but how we can customize it? Let's first stop invenio server.

Open with your favorite editor the `~/.virtualenvs/invenio/var/instance/invenio.cfg`

1. Modify the logo

Let's make our theme beautiful by replacing the logo

in the `~/.virtualenvs/invenio/var/instance/invenio.cfg` add the following:

```
THEME_LOGO = 'images/unicorn.png'
THEME_FRONTPAGE_TITLE = 'Unicorn Institute'
```

Now if you run

```
vagrant> invenio run -h 0.0.0.0
```

and navigate to <http://192.168.50.10> you will see the new logo and front page title.

2. Add facets

Let's replace the facets with the Authors adding the field `main_entry_personal_name.personal_name` in the `~/.virtualenvs/invenio/var/instance/invenio.cfg` add the following:

```
from invenio_records_rest.facets import terms_filter

RECORDS_REST_FACETS = {
    'marc21': {
        'aggs': {
            'author': {
                'terms': {
                    'field': 'main_entry_personal_name.personal_name'
                }
            }
        },
        'post_filters': {
            'author': terms_filter('main_entry_personal_name.personal_name')
```

```
}
}
}
```

Now if you run

```
vagrant> invenio run -h 0.0.0.0
```

and navigate to <http://192.168.50.10/search> you will see that the facets have been replaced with the Authors.

3. Add sort options

in the `~/virtualenvs/invenio/var/instance/invenio.cfg` add the following:

```
RECORDS_REST_SORT_OPTIONS = {
    'records': {
        'title': {
            'fields': ['title_statement.title'],
            'title': 'Record title',
            'order': 1,
        }
    }
}
```

Now if you run

```
vagrant> invenio run -h 0.0.0.0
```

and navigate to <http://192.168.50.10/search> you will see that the sort list have been replaced with the Record title.

4. Change a detail view

We will now replace the template for the detail view of the record, this is possible by changing `RECORDS_UI_ENDPOINTS` with the desired template. In our case we have created the following:

in the `/vagrant/2-customization/invenio-beauty/invenio_beauty/templates/detail.html`

```
{%- extends config.RECORDS_UI_BASE_TEMPLATE %}

{%- macro record_content(data) %}
    {% for key, value in data.items() recursive %}
        <li class="list-group-item">
            {% if value is mapping %}
                <strong>{{ key }}:</strong>
                <ul class="list-group">{{ loop(value.items()) }}</ul>
            {% elif value is iterable and value is not string %}
                <strong>{{ key }}:</strong>
                <ol>
                    {% for item in value %}
                        <li>
                            {% if item is mapping %}
                                <ul class="list-group">
```

```

        {{ record_content(item) }}
    </ul>
    {% else %}
        {{ item }}
    {% endif %}
</li>
{% endfor %}
</ol>
{% else %}
    <strong>{{ key }}:</strong> {{ value }}
{% endif %}
</li>
{% endfor %}
{%- endmacro %}

{%- block page_body %}
<div class="container">
    <div class="row">
        <div class="col-md-12">
            <h2> {{ record.title_statement.title }}</h2>
            <hr />
            <p class="lead">{{ record.summary[0].summary }}</p>
            <hr />
            <h3> {{ _('Metadata') }}</h3>
            <div class="well">
                {{ record_content(record) }}
            </div>
        </div>
    </div>
</div>
{%- endblock %}

```

in the `~/virtualenvs/invenio/var/instance/invenio.cfg` add the following:

```

RECORDS_UI_ENDPOINTS = {
    "recid": {
        "pid_type": "recid",
        "route": "/records/<pid_value>",
        "template": "invenio_beauty/detail.html"
    },
}

```

Now if you run

```
vagrant> invenio run -h 0.0.0.0
```

and navigate to <http://192.168.50.10/records/1> you will see the new template.

5. Modify search results template

We will now replace the search results template, in the search result we are using angular templates and they can easily be configured from the following vars:

- `SEARCH_UI_JSTEMPLATE_COUNT`
- `SEARCH_UI_JSTEMPLATE_ERROR`
- `SEARCH_UI_JSTEMPLATE_FACETS`

- SEARCH_UI_JSTEMPLATE_RANGE
- SEARCH_UI_JSTEMPLATE_LOADING
- SEARCH_UI_JSTEMPLATE_PAGINATION
- SEARCH_UI_JSTEMPLATE_RESULTS
- SEARCH_UI_JSTEMPLATE_SELECT_BOX
- SEARCH_UI_JSTEMPLATE_SORT_ORDER

For our example we will change only SEARCH_UI_JSTEMPLATE_RESULTS, the location of the angular templates are static/templates/<name of your module>

in /vagrant/2-customization/invenio-beauty/invenio_beauty/static/templates/invenio_beauty/results.html

```
<ol>
  <li ng-repeat="record in vm.invenioSearchResults.hits.hits track by $index">
    <span class="label label-success">{{ record.metadata.language_code[0].language_
    ↪code_of_text_sound_track_or_separate_title[0] }}</span>
    <h4><a target="_self" ng-href="/records/{{ record.id }}">{{ record.metadata.title_
    ↪statement.title }}</a></h4>
    <p>{{ record.metadata.summary[0].summary }}</p>
  </li>
</ol>
```

On the angular templates, you have access to the record metadata object, so in you templates you can use {{ record.metadata.foo }}.

Now in the search results template, we will display the language tag on top of each record language_code.

in the ~/.virtualenvs/invenio/var/instance/invenio.cfg add the following:

```
SEARCH_UI_JSTEMPLATE_RESULTS = 'templates/invenio_beauty/results.html'
```

Now if you run

```
vagrant> invenio collect -v
vagrant> invenio run -h 0.0.0.0
```

and navigate to http://192.168.50.10/search you will see the new template.

6. Change the homepage template

We will now replace the demo's homepage. You can change the whole homepage just by replacing THEME_FRONTPAGE_TEMPLATE with your own template, for this example we have created the following:

in /vagrant/2-customization/invenio-beauty/invenio_beauty/templates/invenio_beauty/home.html

```
{%- extends "invenio_theme/page.html" %}
{%- block navbar_search %}{% endblock %}
{%- block page_body %}
  <div class="container">
    <div class="row">
      <div class="col-lg-12">
        <h1 class="text-center">
```



```

    {{_(config.THEME_FRONTPAGE_TITLE)}} Search
  </h1>
  <form action="/search">
    <div class="form-group">
      <input type="text" name="q" class="form-control" placeholder="Type and
↵press enter to search">
    </div>
  </form>
</div>
</div>
</div>
{%- endblock %}

```

If you have a closer look, you will see that we have access to different config variables on the template, by using the config. For example if we want to display the `THEME_FRONTPAGE_TITLE` we can you config. `THEME_FRONTPAGE_TITLE`

So the only thing we should do is to edit the `config.py`

in the `~/virtualenvs/invenio/var/instance/invenio.cfg` add the following:

```
THEME_FRONTPAGE_TEMPLATE = 'invenio_beauty/home.html'
```

Now if you run

```
vagrant> invenio run -h 0.0.0.0
```

and navigate to `http://192.168.50.10` you will see the new template.

Everything together

You want to see the results? Just run the following command.

```
vagrant> cd /vagrant/iugw2017/2-customization
vagrant> cat final.cfg >> ~/virtualenvs/invenio/var/instance/invenio.cfg
```

Running Invenio

Understanding Invenio components

Invenio dems site consists of many components. To see which ones the Invenio demo site uses, you can do:

```

$ pip freeze | grep invenio
invenio-access==1.0.0a11
invenio-accounts==1.0.0b3
invenio-admin==1.0.0b1
invenio-app==1.0.0a1
invenio-app-ils==1.0.0a2
invenio-assets==1.0.0b6
invenio-base==1.0.0a14
invenio-celery==1.0.0b2
invenio-config==1.0.0b3
invenio-db==1.0.0b3
invenio-formatter==1.0.0b1

```

```
invenio-i18n==1.0.0b3
invenio-indexer==1.0.0a9
invenio-jsonschemas==1.0.0a3
invenio-logging==1.0.0b1
invenio-mail==1.0.0b1
invenio-marc21==1.0.0a5
invenio-oaiserver==1.0.0a12
invenio-oauth2server==1.0.0a15
invenio-oauthclient==1.0.0a12
invenio-pidstore==1.0.0b1
invenio-query-parser==0.6.0
invenio-records==1.0.0b1
invenio-records-rest==1.0.0a18
invenio-records-ui==1.0.0a9
invenio-rest==1.0.0a10
invenio-search==1.0.0a9
invenio-search-ui==1.0.0a6
invenio-theme==1.0.0b2
invenio-userprofiles==1.0.0a9
```

Starting the webserver

The Invenio application server can be started using:

```
invenio run -h 0.0.0.0
```

For debugging purposes, you can use:

```
pip install Flask-DebugToolbar
FLASK_DEBUG=1 invenio run --debugger -h 0.0.0.0
```

Starting the job queue

Invenio uses Celery for task execution. The task queue should be started as follows:

```
celery worker -A invenio_app.celery
```

For debugging purposes, you can increase logging level:

```
celery worker -A invenio_app.celery -l DEBUG
```

Using the CLI

Invenio comes with centralised command line. Use `--help` to see available commands:

```
$ invenio --help
Usage: invenio [OPTIONS] COMMAND [ARGS]...

  Command Line Interface for Invenio.

Options:
  --version  Show the flask version
  --help    Show this message and exit.
```

```

Commands:
  access      Account commands.
  alembic     Perform database migrations.
  assets      Web assets commands.
  collect     Collect static files.
  db          Database commands.
  demo        Demo-site commands.
  index       Management command for search indicies.
  instance    Instance commands.
  marc21      MARC21 related commands.
  npm         Generate a package.json file.
  pid         PID-Store management commands.
  records     Record management commands.
  roles       Role commands.
  run         Runs a development server.
  shell       Runs a shell in the app context.
  users       User commands.

```

You can use `--help` for each individual command, for example:

```

$ invenio marc21 import --help
Usage: invenio marc21 import [OPTIONS] INPUT

  Import MARCXML records.

Options:
  --bibliographic
  --authority
  --help           Show this message and exit.

```

Using Python shell

You can start interactive Python shell which will load the Invenio application context so that you can work with the instance:

```

$ invenio shell
Python 2.7.6 (default, Oct 26 2016, 20:30:19)
[GCC 4.8.4] on linux2
App: invenio
Instance: /home/vagrant/.virtualenvs/invenio/var/instance
>>> app.config['BABEL_DEFAULT_LANGUAGE']
'en'
>>> app.config['BROKER_URL']
'amqp://guest:guest@192.168.50.10:5672//'

```

Using administrative interface

You can access administrative interface:

```

$ firefox http://192.168.50.10/admin

```

For example, let us look at the record ID 117 that we have uploaded in *Create and search your first record*. Looking at the administrative interface, we can see that this record has been attributed an internal UUID:

PID_Type	PID	Status	Object Type	Object UUID
oai	oai:invenio:recid/117	REGISTERED	rec	a11dad76-5bd9-471c-975a-0b2b01d74831
recid	117	REGISTERED	rec	a11dad76-5bd9-471c-975a-0b2b01d74831

See *Loading content* for more information about object UUIDs and PIDs.

Loading content

Loading records

In the Invenio demo site example using ILS flavour, we have seen the `invenio marc21` command that can load records directly from a MARCXML file.

You can use `dojson` to convert MARCXML format to its JSON representation:

```
$ dojson -i book.xml -l marcxml do marc21 \
  schema "http://192.168.50.10/schema/marc21/bibliographic/bd-v1.0.0.json" \
  > book.json
$ cat book.json | jq .
[
  {
    "title_statement": {
      "title": "This is title",
      "__order__": [
        "title"
      ]
    },
    "main_entry_personal_name": {
      "personal_name": "Doe, John",
      "__order__": [
        "personal_name"
      ]
    },
    "__order__": [
      "title_statement",
      "main_entry_personal_name"
    ],
    "$schema": "http://192.168.50.10/schema/marc21/bibliographic/bd-v1.0.0.json"
  }
]
```

You can load JSON records using the `invenio records create` command:

```
$ cat book.json | invenio records create --pid-minter recid --pid-minter oaiid
efac2fc2-29af-40bb-a85e-77af0349c0fe
```

The new record that we have just uploaded got the UUID `efac2fc2-29af-40bb-a85e-77af0349c0fe` that uniquely identifies it inside the Invenio record database. It was also minted persistent identifiers `recid` representing record ID and `oaiid` representing OAI ID.

UUIDs and PIDs

Objects managed by Invenio use “internal” UUID identifiers and “external” persistent identifiers (PIDs).

Starting from a persistent identifier, you can see which UUID a persistent identifier points to by using the `invenio pid` command:

```
$ invenio pid get recid 117
rec a11dad76-5bd9-471c-975a-0b2b01d74831 R
```

Starting from the UUID of a record, you can see which PIDs the record was assigned by doing:

```
$ invenio pid dereference rec a11dad76-5bd9-471c-975a-0b2b01d74831
recid 117 None
oai oai:invenio:recid/117 oai
```

You can unassign persistent identifiers:

```
$ invenio pid unassign recid 117
R
$ invenio pid unassign oai oai:invenio:recid/117
R
```

What happens when you try to access the given record ID?

```
$ firefox http://192.168.50.10/api/records/117
```

You can assign another record the same PID:

```
$ invenio pid assign -s REGISTERED -t rec -i 29351009-5e6f-4754-95cb-508f89f4de39_
↪recid 117
```

What happens when you try to access the given record ID now?

```
$ firefox http://192.168.50.10/api/records/117
```

Deleting records

If you want to delete a certain record, you can use:

```
$ invenio records delete -i efac2fc2-29af-40bb-a85e-77af0349c0fe
```

Beware of any registered persistent identifiers, though.

Loading files

Loading full-text files, such as PDF papers or CSV data files together with the records, will be addressed later.

Todo

Describe records, files, buckets.

Accessing content

JSON representation

Records are internally stored in JSON following a certain JSON Schema. They can be obtained using REST API to records.

Using CLI:

```
$ curl -H 'Accept: application/json' http://192.168.50.10/api/records/117
{
  "created": "2017-03-21T10:33:59.245869+00:00",
  "id": 117,
  "links": {
    "self": "http://192.168.50.10/api/records/117"
  },
  "metadata": {
    "__order__": [
      "title_statement",
      "main_entry_personal_name"
    ],
    "_oai": {
      "id": "oai:invenio:recid/117",
      "updated": "2017-03-21T10:33:59Z"
    },
    "control_number": "117",
    "main_entry_personal_name": {
      "__order__": [
        "personal_name"
      ],
      "personal_name": "Doe, John"
    },
    "title_statement": {
      "__order__": [
        "title"
      ],
      "title": "This is title"
    }
  },
  "updated": "2017-03-21T10:33:59.245880+00:00"
}
```

Using Python:

```
$ ipython
In [1]: import requests
In [2]: headers = {'Accept': 'application/json'}
In [3]: r = requests.get('http://192.168.50.10/api/records/117', headers=headers)
In [4]: r.status_code
Out[4]: 200
In [5]: r.json()
Out[5]:
{'created': '2017-03-21T00:01:27.933711+00:00',
 'id': 117,
 'links': {'self': 'http://192.168.50.10/api/records/117'},
 'metadata': {'__order__': ['title_statement', 'main_entry_personal_name'],
 '_oai': {'id': 'oai:invenio:recid/118', 'updated': '2017-03-21T00:01:27Z'},
 'control_number': '118',
```

```
'main_entry_personal_name': {'__order__': ['personal_name'],
 'personal_name': 'Doe, John'},
 'title_statement': {'__order__': ['title'], 'title': 'This is title'}},
 'updated': '2017-03-21T00:01:27.933721+00:00'}
```

Multiple output formats

You can obtain information in other formats by setting an appropriate Accept header. Invenio REST API endpoint will read this information and invoke appropriate record serialisation.

For example, the Invenio demo site runs an ILS flavour and so returns MARCXML by default:

```
$ curl http://192.168.50.10/api/records/117
<?xml version='1.0' encoding='UTF-8'?>
<record xmlns="http://www.loc.gov/MARC21/slim">
  <datafield tag="245" ind1=" " ind2=" ">
    <subfield code="a">This is title</subfield>
  </datafield>
  <datafield tag="100" ind1=" " ind2=" ">
    <subfield code="a">Doe, John</subfield>
  </datafield>
</record>
```

We can ask for Dublin Core:

```
$ curl -H 'Accept: application/xml' http://192.168.50.10/api/records/117
<?xml version='1.0' encoding='UTF-8'?>
<oai_dc:dc xmlns:oai_dc="http://www.openarchives.org/OAI/2.0/oai_dc/" xmlns:xsi=
→ "http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.
→ openarchives.org/OAI/2.0/oai_dc/ http://www.openarchives.org/OAI/2.0/oai_dc.xsd">
  <dc:title xmlns:dc="http://purl.org/dc/elements/1.1/">This is title</dc:title>
  <dc:creator xmlns:dc="http://purl.org/dc/elements/1.1/">Doe, John</dc:creator>
  <dc:type xmlns:dc="http://purl.org/dc/elements/1.1/">
  <dc:language xmlns:dc="http://purl.org/dc/elements/1.1/">
</oai_dc:dc>
```

Getting record fields

Getting title

If we would like to obtain only some part of information, for example record title, we can simply filter the output fields.

Using CLI:

```
$ curl -s -H 'Accept: application/json' http://192.168.50.10/api/records/117 | \
jq -r '.metadata.title_statement.title'
This is title
```

Using Python:

```
$ ipython
In [1]: import requests
In [2]: headers = {'Accept': 'application/json'}
In [3]: r = requests.get('http://192.168.50.10/api/records/117', headers=headers)
```

```
In [4]: r.json()['metadata'].get('title_statement', {}).get('title', '')
Out[4]: 'This is title'
```

Getting co-authors

If we would like to print all co-author names, we can iterate over respective JSON field as follows:

Using CLI:

```
$ curl -s -H 'Accept: application/json' http://192.168.50.10/api/records/97 | \
jq -r '.metadata.added_entry_personal_name[].personal_name'
Lokajczyk, T
Xu, W
Jastrow, U
Hahn, U
Bittner, L
Feldhaus, J
```

Using Python:

```
$ ipython
In [1]: import requests
In [2]: headers = {'Accept': 'application/json'}
In [3]: r = requests.get('http://192.168.50.10/api/records/97', headers=headers)
In [3]: for coauthor in r.json()['metadata']['added_entry_personal_name']:
.....:     print(coauthor['personal_name'])
Lokajczyk, T
Xu, W
Jastrow, U
Hahn, U
Bittner, L
Feldhaus, J
```

Searching records

Invenio instance can be searched programmatically via the REST API endpoint:

```
$ curl -H 'Accept: application/json' http://192.168.50.10/api/records?q=model
```

Note the pagination of the output done by the “links” output field.

How many records are there that contain the word “model”? We need to iterate over results:

```
nb_hits = 0

def get_nb_hits(json_response):
    return len(json_response['hits']['hits'])

def get_next_link(json_response):
    return json_response['links'].get('next', None)

response = requests.get('http://192.168.50.10/api/records?q=model', headers=headers).
↳ json()
nb_hits += get_nb_hits(response)
while get_next_link(response):
```



```
response = requests.get(get_next_link(response), headers=headers).json()
nb_hits += get_nb_hits(response)

print(nb_hits)
```

Enabling ORCID login

ORCID provides a persistent digital identifier that distinguishes you from every other researcher and, through integration in key research workflows such as manuscript and grant submission, supports automated linkages between you and your professional activities ensuring that your work is recognized.

Say that your institutional library or repository has decided to enable ORCID integration, so that:

- user can login into your system through ORCID, thus being identified directly through their very personal and verified ORCID Id.
- the service can *pull from* or *push to* information related to the user and on behalf of the user, directly with ORCID

In this tutorial will be present the former point, while the latter depends on the `invenio-orcid` module that is currently subject to important changes.

Invenio support out-of-the-box authentication through the **OAUTH** protocol, which is actually used by the ORCID service to offer authentication.

Registering an ORCID member API client application

In order to integrate your Invenio instance with ORCID the first step is to [apply for API keys](#) to access the ORCID Sandbox.

Please, follow the official ORCID documentation for this part.

After successful application for an API client application key you should receive in your inbox an email with your `Client ID` and `Client secret`. You will need this information when configuring Invenio in the next step.

Notes on the redirect URI

As part of the OAUTH authentication process, after the user will have authenticated on the ORCID site, the user will be redirected to a given page on the Invenio side. ORCID requires to provide a list of authorized URI prefixes that could be allowed for this redirection to happen.

Depending on the `SERVER_NAME` used to configure the Invenio installation, you should fill this parameter with:

```
https://<SERVER_NAME>/oauth/authorized/orcid/
```

Enabling OAUTH+ORCID in your Invenio instance

In order to enable OAUTH authentication for ORCID, just add these line to your `var/instance/invenio.cfg` file.

```
from invenio_oauthclient.contrib import orcid

OAUTHCLIENT_REMOTE_APPS = dict(
    orcid=orcid.REMOTE_SANDBOX_APP,
```

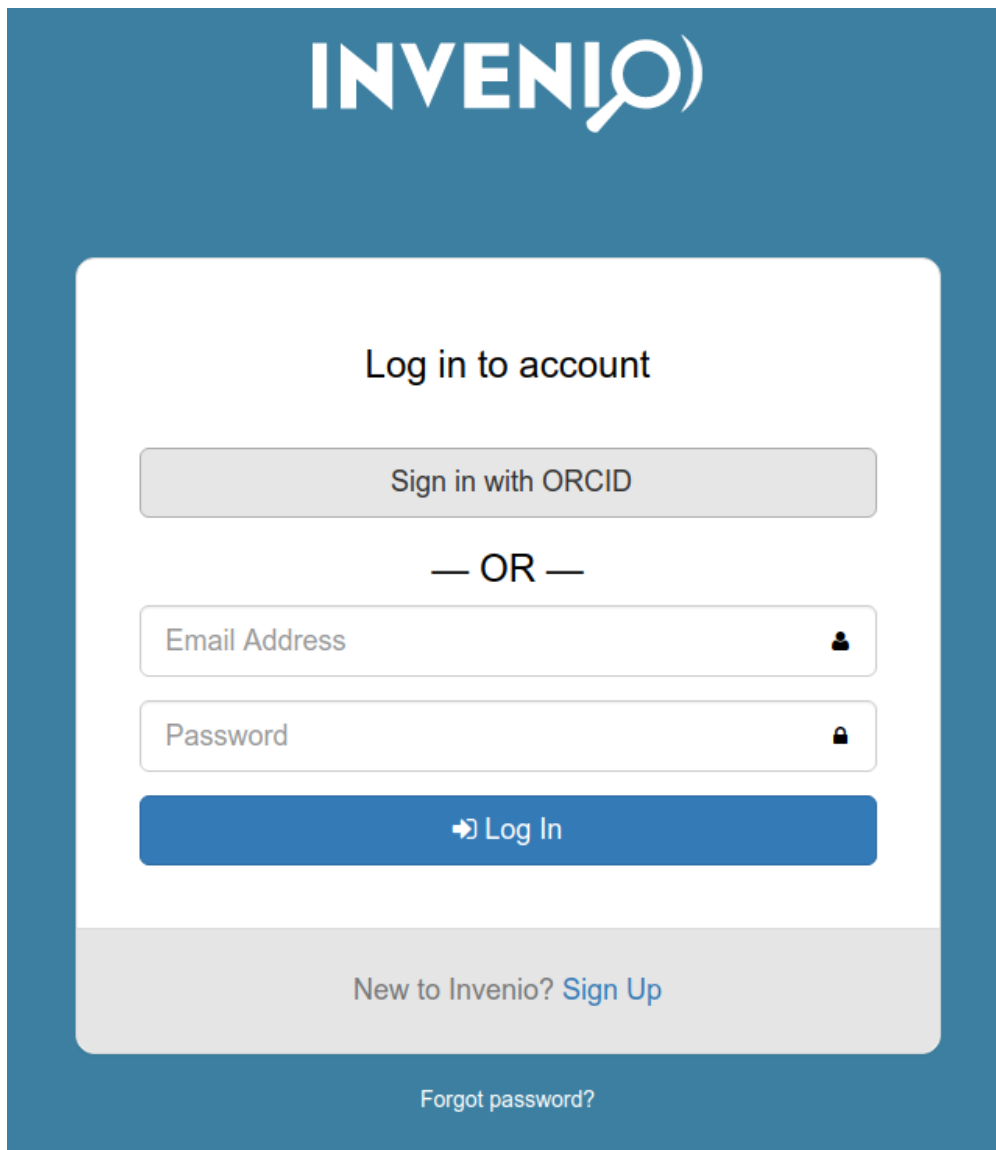
```
)  
  
ORCID_APP_CREDENTIALS = dict(  
    consumer_key="Client ID",  
    consumer_secret="Client secret",  
)
```

where the `Client ID` and `Client secret` are those provided by ORCID itself in the previous step.

If you now visit:

```
https://<SERVER_NAME>/login
```

you will be able to see ORCID authentication enabled:





INVENIO

Log in to account

Sign in with ORCID

— OR —

Email Address 

Password 

Log In

New to Invenio? [Sign Up](#)

[Forgot password?](#)

When a user land on Invenio after having passed the ORCID authentication phase, Invenio will not be provided by ORCID with the user email address. For this reason, upon the first time the user logs-in into your instance, the user will be asked to fill in her email address that will have subsequently to be confirmed.

Notes on setting up the OAUTH scope

In OAUTH, the user actually authorize the given service (i.e. your Invenio installation) to act on behalf of the user within a certain *scope*. By default, ORCID will authorize Invenio to only know the ORCID ID of the authenticating user.

If you plan to build more advanced services, such as pushing and pulling information with ORCID, you might wish to enable broader OAUTH scopes, so that, upon authentication, your Invenio instance is actually able to perform them.

E.g. the INSPIRE service is currently using this scope configuration in their `invenio.cfg`:

```
orcid.REMOTE_SANDBOX_APP['params']['request_token_params'] = {
    'scope': '/orcid-profile/read-limited '
            '/activities/update /orcid-bio/update'
}
```

Upon login with your Invenio instance through ORCID, your users will be presented then with a screen similar to the following one:



Invenio User Group Workshop 2017 ORCID Hands-on session ?

has asked for the following access to your ORCID Record



Add or update your research activities

Add or update your biographical information

Read your ORCID record

- Allow this permission until I revoke it.
*You may revoke permissions on your account settings page.
 Unchecking this box will grant permission this time only.*

This application will not be able to see your ORCID password, or other private info in your ORCID Record. [Privacy Policy](#).

Where to go from here?

In this tutorial, we have presented how to integrate ORCID authentication into your Invenio instance.

As a developer, you will be able to extract the ORCID ID of the authenticating user from her user by querying the `RemoteAccount` table.

For exchanging information with ORCID, this highly depends on the data model implemented in your instance and what type of information you plan to exchange with ORCID.

`invenio-orcid` module is currently being developed in order to make it easier for Invenio instances to build information exchange with ORCID.

Migrating to v3

Dump Invenio v1.2 or v2.1 data

Dump CLI installation

There are several ways of installing `invenio-migrator` in your Invenio v1.2 or v2.1 production environment, the one we recommend is using `Virtualenv` to avoid any interference with the currently installed libraries:

```
$ sudo pip install virtualenv virtualenvwrapper
$ source /usr/local/bin/virtualenvwrapper.sh
$ mkvirtualenv migration --system-site-packages
$ workon migration
$ pip install invenio-migrator --pre
$ inveniomigrator dump --help
```

It is important to use the option `--system-site-packages` as `Invenio-Migrator` will use Invenio legacy python APIs to perform the dump. The package `virtualenvwrapper` is not required but it is quite convenient.

Dump records and files

```
$ mkdir /vagrant/dump
$ cd /vagrant/dump/
$ inveniomigrator dump records
```

This will generate one or more JSON files containing 1000 records each tops, with the following information:

- The record id.
- The record metadata, stored in the `record` key there is a list with one item for each of the revisions of the record, and each item of the list contains the MARC21 representation of the record plus the optional JSON.
- The files linked with the record, like for the record metadata it is a list with all the revisions of the files.
- Optionally it also contains the collections the record belongs to.

For more information about how to dump records and files see the [Usage section](#) of the `Invenio-Migrator` documentation.

The file path inside the Invenio legacy installation will be included in the dump and used as file location for the new Invenio v3 installation. If you are able to mount the file system following the same pattern in your Invenio v3 machines, there shouldn't be any problem, but if you can't do it, then you need to copy over the files folder manually using your favorite method, i.e.:

```
$ cd /opt/invenio/var/data
$ tar -zcvf /vagrant/dump/files.tar.gz files
```

Pro-tip: Maybe you want to have different data models in your new installation depending on the nature of the record, i.e. bibliographic records vs authority records. In this case one option is to dump them in different files using the `--query` argument when dumping from your legacy installation:

```
$ inveniomigrator dump records --query '-980__a:AUTHORITY' --file-prefix bib
$ inveniomigrator dump records --query '980__a:AUTHORITY' --file-prefix auth
```

Things

The dump command of the Invenio-Migrator works with, what we called, *things*. A *thing* is an entity you want to dump from Invenio legacy, in the previous example the *thing* was records. The list of *things* Invenio-Migrator can dump by default is listed via entry-points in the `setup.py`, this not only help us add new dump scripts easily, but also allows anyone to create their own from outside the Invenio-Migrator.

You can read more about which *things* are already supported by the [Invenio-Migrator documentation](#) and also we have section dedicated to [Entry points](#) where you can learn how we used them to extend Invenio's functionality.

Load data into Invenio v3

Load CLI installation

Invenio-Migrator can be installed in any Invenio 3 environment using PyPi and the extra dependencies loader:

```
$ pip install invenio-migrator[loader]
```

Depending on what you want to load you might need to have installed other packages, i.e. to load communities from Invenio 2.1 `invenio-communities` needs to be installed.

This will add to your Invenio application a new set of commands under `dumps`:

```
$ invenio dumps --help
```

Load records and files

```
$ invenio dumps loadrecords /vagrant/dump/records_dump_0.json
```

This will generate one celery task to import each of the records inside the dump.

Pro-tip: By default Invenio-Migrator uses the bibliographic MARC21 standard to transform and load the records, we now that this might not be the case to all Invenio v3 installation, i.e. authority records. By changing `MIGRATOR_RECORDS_DUMP_CLS` and `MIGRATOR_RECORDS_DUMPLOADER_CLS` you can customize the behavior of the loading command. There is a full chapter in the [Invenio-Migrator documentation](#) about [customizing loading](#) if you want more information.

Loaders

Each of the entities that can be loaded by Invenio-Migrator have a companion command generally prefixed by *load*, i.e. `loadrecords`.

The loaders are similar to the things we describe previously, but in this case, instead of entry-points, if you want to extend the default list of loaders it can be done adding a new command to `dumps`, more information about the loaders can be found in the [Invenio-Migrator documentation](#) and on how to add more commands in the [click documentation](#).

This part of the documentation will show you how to setup a production system for running Invenio.

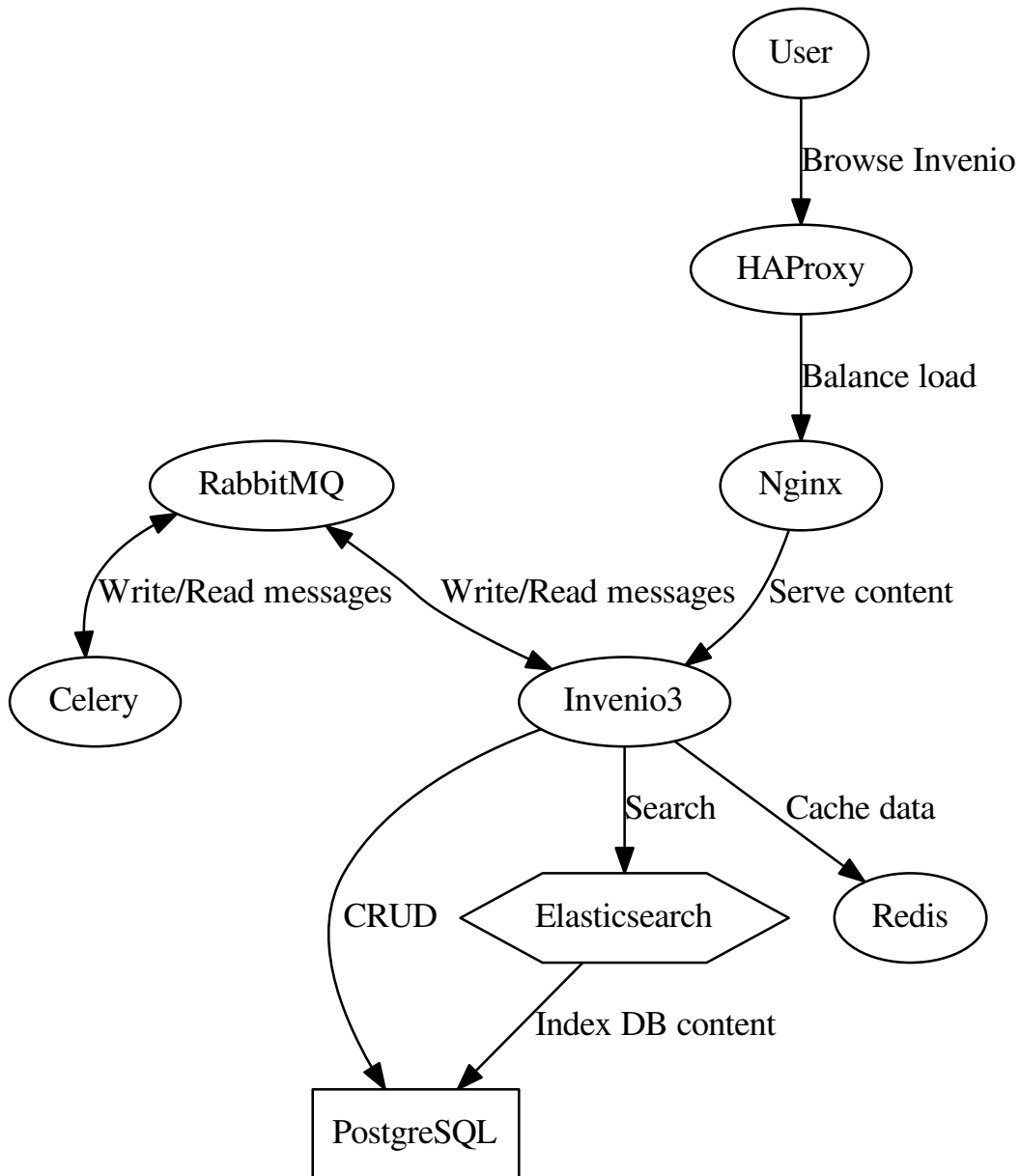
Overview

Services

In this documentation we will provide you an overview of the different services needed to run invenio. We will explain what are their roles in the global architecture, and why they have been selected.

Invenio 3 has been designed to be highly scalable, low latency, and fast, therefore all the underlying services need to be respecting the same goals.

Here is an overview of the system:



Elasticsearch

Invenio 3 is an external search engine service. It will take care of everything related to the search as indexing, ranking, searching. It contains and ingest document under the form of JSON, but it is able to respect type. To be able to determine the data structure that you want to ingests and the type of the containing information, it requires you to create a mapping.

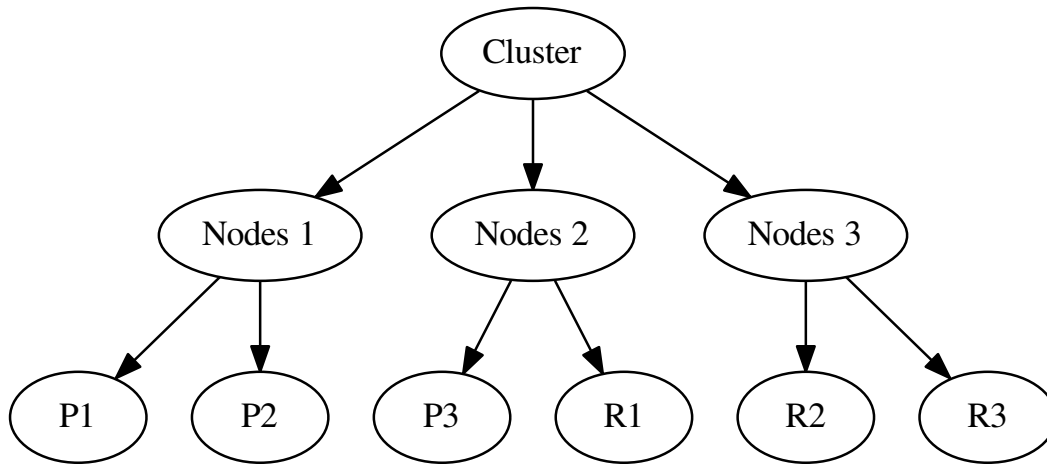
A **mapping** is a simple JSON structure which describes how should look an indexed document.

For example:

```
{
  "user" : {
    "properties" : {
      "name" : {
        "properties" : {
          "first" : {
            "type" : "string"
          }
        }
      }
    }
  }
}
```

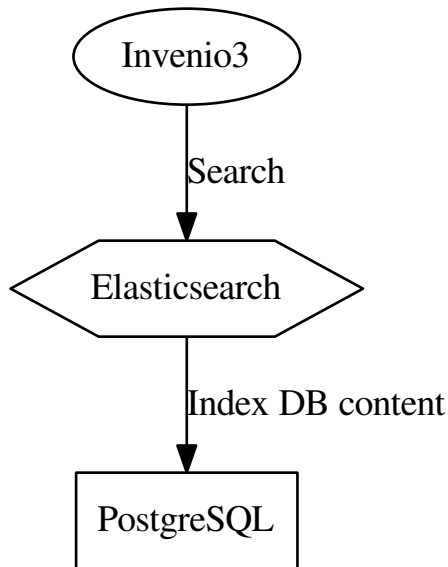
You can basically store in Elasticsearch almost everything, so it also means that you can make searchable almost any data on your Invenio 3 instances.

Elastic search is highly scalable, which means that it can spread the load on several instances of elasticsearch. In production you will always try to have a dedicated Elasticsearch cluster. Indeed, without it, it is not revealing its full power. Each node is running several instances of Elasticsearch, this unit is named “Shards” it can either be replica instances or production ones.



This allow Invenio 3 to handle really large amount of records and keep the search time low. You can find more information [here](#).

Summary of elasticsearch position in Invenio3:



PostgreSQL

PostgreSQL is currently the database selected for Invenio 3, it is replacing mysql/mariadb in Invenio 1, the version must be 9.4 or newer.

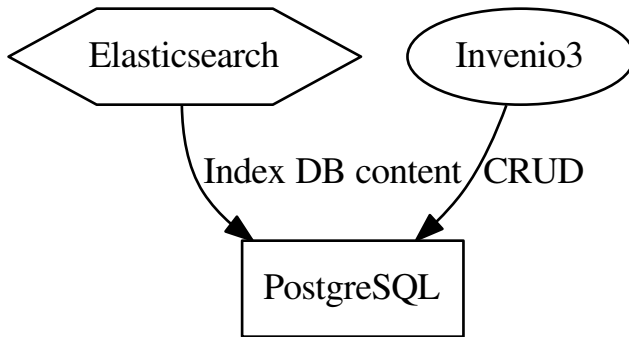
Elasticsearch is used to index and search data, but it is not here to persistently store data. It also doesn't include a transactional system to manage concurrent requests. We use a database to store in a persistent way data, Elasticsearch would feed from the database, before to be able to answer search requests.

PostgreSQL has been selected for its really high performances, even if it's not really scalable the transactions here will be mostly "write" operation, most of the read operation will be delegated to Elasticsearch.

PostgreSQL is the perfect fit for the Invenio 3, indeed, a everything is JSON, from the mappings to the schema and documents, and this database has all the necessary feature to be able to handle this data type efficiently. Most of the data are not represented as usual where we have a column per field. In the case of Invenio 3 the documents are JSON stored as JSON objects. Even if it not composed of column PostgreSQL is able to do operation of this kind of object.

PostgreSQL is abstracted in Invenio 3 code thanks to the use of the framework SQLAlchemy, it means that you don't need to know how to use PostgreSQL but python would be enough.

Summary of PostgreSQL in Invenio 3:



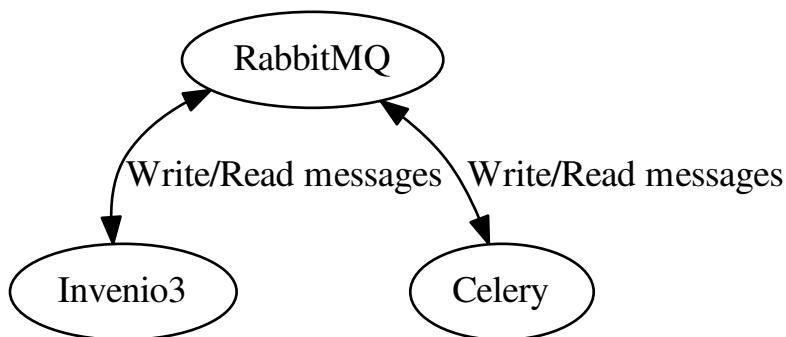
RabbitMQ

RabbitMQ is a messaging queue service which is used to make different processes communicating between each others by letting them exchange messages.

RabbitMQ is highly scalable and can make processes communicate between several nodes, it uses a system of broker to transmit the messages between the applications. It can handle a lot of messages in a fast and efficient way.

In the case of Invenio 3 the messaging queue is used to transmit messages between Invenio and Celery nodes.

Summary of RabbitMQ in Invenio 3:



Celery/Flower

Celery is an asynchronous task queue, that can also act as a scheduler for recurring tasks. In our case the tasks are transmitted thanks to RabbitMQ under the form of messages. Celery is reading in the message queue which tasks need to be executed, then it execute it, and write the result back in the queue.

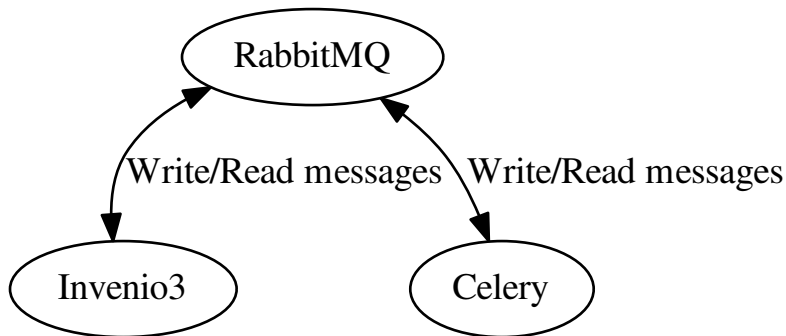
Celery in Invenio 3 is used in different cases:

- The first one is for heavy process, we can't let a user hanging for a long time. So when we have an operation that should take a long time to execute it is sent to Celery to be executed as soon as possible.
- The second one is for recurring tasks, it replaces BibSched in Invenio 1. Different modules in Invenio 3 can register tasks that will be executed when needed. An example can be the harvesting of some records.

Celery is working with RabbitMQ and can be highly scalable, the idea is that you can have as many computing nodes running celery connected to the messaging queue. It is then really easy to add more nodes if the load is too high.

It can be hard to know what is running in Celery which tasks did succeed and which one failed, therefore there is a tool that can help to monitor what is happening. It is named **Flower** it takes the form of a website that gives you an overview of what is happening.

Summary of Celery in Invenio 3:



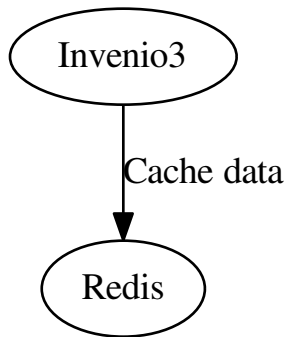
Redis

Redis is a key value service that allows to store information that need to be retrieved with a really high access speed. It can be used to cache data, or as a messaging queue like RabbitMQ, it is currently possible to communicate with celery thanks to Redis instead of RabbitMQ.

In Invenio we mostly use it for caching data, and example is to cache the user session, it is way faster to store the data in Redis than in the database. Even if Redis can have some persistency we would prefer the database to store such data.

Redis is again a service which is really scalable it is possible to have it on separated nodes that will be dedicated to it. It can be really helpful as Redis will have a high consumption in memory, but really small need in terms of computing power.

Summary of Redis in Invenio 3:



Nginx

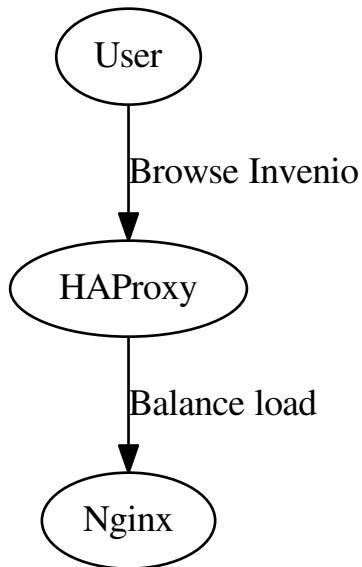
Nginx is a webserver that is extremely efficient for serving static files. It is used as a reverse proxy between the user and Invenio 3. It adds some logic and features linked to the connexion handling and the distribution of the requests. For example nginx can handle DDOS attacks.

Nginx will make the link between the front end of Invenio 3 that will be served as static files when possible and the RESTFUL api behind.

HAProxy

HAProxy is a load balancer that will distribute HTTP requests amongst several servers. It is not mandatory, but it can be really useful if you have a really high traffic website. The idea is to spread the load to several webserver. This way we can avoid the saturation and then the unavailability of the webserver.

Summary of Nginx and HAProxy in Invenio 3:



Detailed installation guide

CAVEAT LECTOR

Invenio v3.0 alpha is a bleeding-edge developer preview version.

Introduction

In this installation guide, we’ll create an Invenio digital library instance using a multi-machine setup where separate services (such as the database server and the web server) run on separate dedicated machines. Such a multi-machine setup emulates to what one would typically use in production. (However, it is very well possible to follow this guide and install all the services onto the same “localhost”, if one wants to.)

We’ll use six dedicated machines running the following services:

node	IP	runs
web	192.168.50.10	Invenio web application
postgresql	192.168.50.11	PostgreSQL database server
redis	192.168.50.12	Redis caching service
elasticsearch	192.168.50.13	Elasticsearch information retrieval service
rabbitmq	192.168.50.14	RabbitMQ messaging service
worker	192.168.50.15	Celery worker node

The instructions below are tested on Ubuntu 14.04 LTS (Trusty Tahr) and CentOS 7 operating systems. For other operating systems such as Mac OS X, you may want to check out the “kickstart” set of scripts coming with the Invenio source code that perform the below-quoted installation steps in an unattended automated way.

Environment variables

Let's define some useful environment variables that will describe our Invenio instance setup:

INVENIO_WEB_HOST The IP address of the Web server node.

INVENIO_WEB_INSTANCE The name of your Invenio instance that will be created. Usually equal to the name of the Python virtual environment.

INVENIO_WEB_VENV The name of the Python virtual environment where Invenio will be installed. Usually equal to the name of the Invenio instance.

INVENIO_USER_EMAIL The email address of a user account that will be created on the Invenio instance.

INVENIO_USER_PASS The password of this Invenio user.

INVENIO_POSTGRESQL_HOST The IP address of the PostgreSQL database server.

INVENIO_POSTGRESQL_DBNAME The database name that will hold persistent data of our Invenio instance.

INVENIO_POSTGRESQL_DBUSER The database user name used to connect to the database server.

INVENIO_POSTGRESQL_DBPASS The password of this database user.

INVENIO_REDIS_HOST The IP address of the Redis server.

INVENIO_ELASTICSEARCH_HOST The IP address of the Elasticsearch information retrieval server.

INVENIO_RABBITMQ_HOST The IP address of the RabbitMQ messaging server.

INVENIO_WORKER_HOST The IP address of the Celery worker node.

In our example setup, we shall use:

```
export INVENIO_WEB_HOST=192.168.50.10
export INVENIO_WEB_INSTANCE=invenio
export INVENIO_WEB_VENV=invenio
export INVENIO_USER_EMAIL=info@inveniosoftware.org
export INVENIO_USER_PASS=uspass123
export INVENIO_POSTGRESQL_HOST=192.168.50.11
export INVENIO_POSTGRESQL_DBNAME=invenio
export INVENIO_POSTGRESQL_DBUSER=invenio
export INVENIO_POSTGRESQL_DBPASS=dbpass123
export INVENIO_REDIS_HOST=192.168.50.12
export INVENIO_ELASTICSEARCH_HOST=192.168.50.13
export INVENIO_RABBITMQ_HOST=192.168.50.14
export INVENIO_WORKER_HOST=192.168.50.15
```

Let us save this configuration in a file called `.inveniorc` for future use.

Web

The web application node (192.168.50.10) is where the main Invenio application will be running. We need to provision it with some system dependencies in order to be able to install various underlying Python and JavaScript libraries.

The web application node can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-web.sh
```

Let's see in detail what the web provisioning script does.

First, let's see if using `sudo` will be required:

```
# runs as root or needs sudo?
if [[ "$EUID" -ne 0 ]]; then
    sudo='sudo'
else
    sudo=''
fi
```

Second, some useful system tools are installed:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
# update list of available packages:
$sudo apt-get -y update

# install useful system tools:
$sudo apt-get -y install \
    curl \
    git \
    rlwrap \
    screen \
    vim
```

- on CentOS 7:

```
# add EPEL external repository:
$sudo yum install -y epel-release

# install useful system tools:
$sudo yum install -y \
    curl \
    git \
    rlwrap \
    screen \
    vim
```

Third, an external Node.js package repository is enabled. We'll be needing to install and run Npm on the web node later. The Node.js repository is enabled as follows:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
if [[ ! -f /etc/apt/sources.list.d/nodesource.list ]]; then
    curl -sL https://deb.nodesource.com/setup_4.x | $sudo bash -
fi
```

- on CentOS 7:

```
if [[ ! -f /etc/yum.repos.d/nodesource-el.repo ]]; then
```



```
curl -sL https://rpm.nodesource.com/setup_4.x | $sudo bash -
fi
```

Fourth, all the common prerequisite software libraries and packages that Invenio needs are installed:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
$sudo apt-get -y install \
  libffi-dev \
  libfreetype6-dev \
  libjpeg-dev \
  libmsgpack-dev \
  libssl-dev \
  libtiff-dev \
  libxml2-dev \
  libxslt-dev \
  nodejs \
  python-dev \
  python-pip
```

- on CentOS7:

```
# install development tools:
$sudo yum update -y
$sudo yum groupinstall -y "Development Tools"
$sudo yum install -y \
  libffi-devel \
  libxml2-devel \
  libxslt-devel \
  openssl-devel \
  policycoreutils-python \
  python-devel \
  python-pip
$sudo yum install -y --disablerepo=epel \
  nodejs
```

We want to use PostgreSQL database in this installation example, so we need to install corresponding libraries too:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
$sudo apt-get -y install \
  libpq-dev
```

- on CentOS7:

```
$sudo yum install -y \
  postgresql-devel
```

Fifth, now that Node.js is installed, we can proceed with installing Npm and associated CSS/JS filter tools. Let's do it globally:

- on either of the operating systems:

```
# $sudo su -c "npm install -g npm"
$sudo su -c "npm install -g node-sass@3.8.0 clean-css@3.4.12 requirejs uglify-js"
```

Sixth, we'll install Python virtual environment wrapper tools and activate them in the current user shell process:

- on either of the operating systems:

```
$sudo pip install -U setuptools pip
$sudo pip install -U virtualenvwrapper
if ! grep -q virtualenvwrapper ~/.bashrc; then
    mkdir -p "$HOME/.virtualenvs"
    echo "export WORKON_HOME=$HOME/.virtualenvs" >> "$HOME/.bashrc"
    echo "source $(which virtualenvwrapper.sh)" >> "$HOME/.bashrc"
fi
export WORKON_HOME=$HOME/.virtualenvs
# shellcheck source=/dev/null
source "$(which virtualenvwrapper.sh)"
```

Seventh, we install Nginx web server and configure appropriate virtual host:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
# install Nginx web server:
$sudo apt-get install -y nginx

# configure Nginx web server:
$sudo cp -f "$scriptpathname/../nginx/invenio.conf" /etc/nginx/sites-available/
$sudo sed -i "s,/home/invenio/,/home/${whoami}/,g" /etc/nginx/sites-available/
↪invenio.conf
$sudo rm /etc/nginx/sites-enabled/default
$sudo ln -s /etc/nginx/sites-available/invenio.conf /etc/nginx/sites-enabled/
$sudo /usr/sbin/service nginx restart
```

- on CentOS7:

```
# install Nginx web server:
$sudo yum install -y nginx

# configure Nginx web server:
$sudo cp "$scriptpathname/../nginx/invenio.conf" /etc/nginx/conf.d/
$sudo sed -i "s,/home/invenio/,/home/${whoami}/,g" /etc/nginx/conf.d/invenio.conf

# add SELinux permissions if necessary:
if $sudo getenforce | grep -q 'Enforcing'; then
    if ! $sudo semanage port -l | tail -n +1 | grep -q '8888'; then
        $sudo semanage port -a -t http_port_t -p tcp 8888
    fi
    if ! $sudo semanage port -l | grep ^http_port_t | grep -q 5000; then
        $sudo semanage port -m -t http_port_t -p tcp 5000
    fi
fi
```

```

        if ! $sudo getsebool -a | grep httpd | grep httpd_enable_homedirs | grep -q
↪on; then
            $sudo setsebool -P httpd_enable_homedirs 1
            mkdir -p "/home/${whoami}/.virtualenvs/${INVENIO_WEB_VENV}/var/instance/
↪static"
            $sudo chcon -R -t httpd_sys_content_t "/home/${whoami}/.virtualenvs/$
↪{INVENIO_WEB_VENV}/var/instance/static"
        fi
    fi

    $sudo sed -i 's,80,8888,g' /etc/nginx/nginx.conf
    $sudo chmod go+rx "/home/${whoami}/"
    $sudo /sbin/service nginx restart

    # open firewall ports:
    if firewall-cmd --state | grep -q running; then
        $sudo firewall-cmd --permanent --zone=public --add-service=http
        $sudo firewall-cmd --permanent --zone=public --add-service=https
        $sudo firewall-cmd --reload
    fi

```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
$sudo apt-get -y autoremove && $sudo apt-get -y clean
```

- on CentOS7:

```
$sudo yum clean -y all
```

Database

The database server (192.168.50.11) will hold persistent data of our Invenio installation, such as bibliographic records or user accounts. Invenio supports MySQL, PostgreSQL, and SQLite databases. In this tutorial, we shall use PostgreSQL that is the recommended database platform for Invenio.

The database server node can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-postgresql.sh
```

Let's see in detail what the database provisioning script does.

First, we install and configure the database software:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
# update list of available packages:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y update
```

```

# install PostgreSQL:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y install \
    postgresql

# allow network connections:
if ! grep -q "listen_addresses.*${INVENIO_POSTGRESQL_HOST}" \
    /etc/postgresql/9.3/main/postgresql.conf; then
    echo "listen_addresses = '${INVENIO_POSTGRESQL_HOST}'" | \
        sudo tee -a /etc/postgresql/9.3/main/postgresql.conf
fi

# grant access rights:
if ! sudo grep -q "host.*${INVENIO_POSTGRESQL_DBNAME}.*${INVENIO_POSTGRESQL_
↳DBUSER}" \
    /etc/postgresql/9.3/main/pg_hba.conf; then
    echo "host ${INVENIO_POSTGRESQL_DBNAME} ${INVENIO_POSTGRESQL_DBUSER} $
↳{INVENIO_WEB_HOST}/32 md5" | \
        sudo tee -a /etc/postgresql/9.3/main/pg_hba.conf
fi

# grant database creation rights via SQLAlchemy-Utills:
if ! sudo grep -q "host.*templatel.*${INVENIO_POSTGRESQL_DBUSER}" \
    /etc/postgresql/9.3/main/pg_hba.conf; then
    echo "host templatel ${INVENIO_POSTGRESQL_DBUSER} ${INVENIO_WEB_HOST}/32 md5"
↳| \
        sudo tee -a /etc/postgresql/9.3/main/pg_hba.conf
fi

# restart PostgreSQL server:
sudo /etc/init.d/postgresql restart

```

- on CentOS 7:

```

# add EPEL external repository:
sudo yum install -y epel-release

# install PostgreSQL:
sudo yum update -y
sudo yum install -y \
    postgresql-server

# initialise PostgreSQL database:
sudo -i -u postgres pg_ctl initdb

# allow network connections:
if ! sudo grep -q "listen_addresses.*${INVENIO_POSTGRESQL_HOST}" \
    /var/lib/pgsql/data/postgresql.conf; then
    echo "listen_addresses = '${INVENIO_POSTGRESQL_HOST}'" | \
        sudo tee -a /var/lib/pgsql/data/postgresql.conf
fi

# grant access rights:
if ! sudo grep -q "host.*${INVENIO_POSTGRESQL_DBNAME}.*${INVENIO_POSTGRESQL_
↳DBUSER}" \
    /var/lib/pgsql/data/pg_hba.conf; then
    echo "host ${INVENIO_POSTGRESQL_DBNAME} ${INVENIO_POSTGRESQL_DBUSER} $
↳{INVENIO_WEB_HOST}/32 md5" | \

```

```

        sudo tee -a /var/lib/pgsql/data/pg_hba.conf
fi

# grant database creation rights via SQLAlchemy-Utils:
if ! sudo grep -q "host.*template1.*${INVENIO_POSTGRESQL_DBUSER}" \
    /var/lib/pgsql/data/pg_hba.conf; then
    echo "host template1 ${INVENIO_POSTGRESQL_DBUSER} ${INVENIO_WEB_HOST}/32 md5"
↪ | \
        sudo tee -a /var/lib/pgsql/data/pg_hba.conf
fi

# open firewall ports:
if firewall-cmd --state | grep -q running; then
    sudo firewall-cmd --zone=public --add-service=postgresql --permanent
    sudo firewall-cmd --reload
fi

# enable PostgreSQL upon reboot:
sudo systemctl enable postgresql

# restart PostgreSQL server:
sudo systemctl start postgresql

```

We can now create a new database user with the necessary access permissions on the new database:

- on either of the operating systems:

```

# create user if it does not exist:
echo "SELECT 1 FROM pg_roles WHERE rolname='${INVENIO_POSTGRESQL_DBUSER}'" | \
    sudo -i -u postgres psql -tA | grep -q 1 || \
    echo "CREATE USER ${INVENIO_POSTGRESQL_DBUSER} WITH PASSWORD '${INVENIO_
↪ POSTGRESQL_DBPASS}';" | \
        sudo -i -u postgres psql

# create database if it does not exist:
echo "SELECT 1 FROM pg_database WHERE datname='${INVENIO_POSTGRESQL_DBNAME}'" | \
    sudo -i -u postgres psql -tA | grep -q 1 || \
    echo "CREATE DATABASE ${INVENIO_POSTGRESQL_DBNAME};" | \
        sudo -i -u postgres psql

# grant privileges to the user on this database:
echo "GRANT ALL PRIVILEGES ON DATABASE ${INVENIO_POSTGRESQL_DBNAME} TO ${INVENIO_
↪ POSTGRESQL_DBUSER};" | \
        sudo -i -u postgres psql

```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
sudo apt-get -y autoremove && sudo apt-get -y clean
```

- on CentOS7:

```
sudo yum clean -y all
```

Redis

The Redis server (192.168.50.12) is used for various caching needs.

The Redis server can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-redis.sh
```

Let's see in detail what the Redis provisioning script does.

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
# update list of available packages:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y update

# install Redis server:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y install \
    redis-server

# allow network connections:
if ! grep -q "${INVENIO_REDIS_HOST}" /etc/redis/redis.conf; then
    sudo sed -i "s/bind 127.0.0.1/bind 127.0.0.1 ${INVENIO_REDIS_HOST}/" \
        /etc/redis/redis.conf
fi

# restart Redis server:
sudo /etc/init.d/redis-server restart
```

- on CentOS 7:

```
# add EPEL external repository:
sudo yum install -y epel-release

# update list of available packages:
sudo yum update -y

# install Redis server:
sudo yum install -y \
    redis

# allow network connections:
if ! grep -q "${INVENIO_REDIS_HOST}" /etc/redis.conf; then
    sudo sed -i "s/bind 127.0.0.1/bind 127.0.0.1 ${INVENIO_REDIS_HOST}/" \
        /etc/redis.conf
fi

# open firewall ports:
if firewall-cmd --state | grep -q running; then
    sudo firewall-cmd --zone=public --add-port=6379/tcp --permanent
```

```

sudo firewall-cmd --reload
fi

# enable Redis upon reboot:
sudo systemctl enable redis

# start Redis:
sudo systemctl start redis

```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
sudo apt-get -y autoremove && sudo apt-get -y clean
```

- on CentOS7:

```
sudo yum clean -y all
```

Elasticsearch

The Elasticsearch server (192.168.50.13) is used to index and search bibliographic records, fulltext documents, and other various interesting information managed by our Invenio digital library instance.

The Elasticsearch server can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-elasticsearch.sh
```

Let's see in detail what the Elasticsearch provisioning script does.

- on Ubuntu 14.04 LTS (Trusty Tahr):

```

# install curl:
sudo apt-get -y install curl

# add external Elasticsearch repository:
if [[ ! -f /etc/apt/sources.list.d/elasticsearch-2.x.list ]]; then
  curl -sL https://packages.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add
  ↪-
  echo "deb http://packages.elastic.co/elasticsearch/2.x/debian stable main" | \
    sudo tee -a /etc/apt/sources.list.d/elasticsearch-2.x.list
fi

# update list of available packages:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y update

# install Elasticsearch server:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y install \
  elasticsearch \
  openjdk-7-jre

```

```
# allow network connections:
if ! sudo grep -q "network.host: ${INVENIO_ELASTICSEARCH_HOST}" \
    /etc/elasticsearch/elasticsearch.yml; then
    echo "network.host: ${INVENIO_ELASTICSEARCH_HOST}" | \
        sudo tee -a /etc/elasticsearch/elasticsearch.yml
fi

# enable Elasticsearch upon reboot:
sudo update-rc.d elasticsearch defaults 95 10

# start Elasticsearch:
sudo /etc/init.d/elasticsearch restart
```

- on CentOS 7:

```
# add external Elasticsearch repository:
if [[ ! -f /etc/yum.repos.d/elasticsearch.repo ]]; then
    sudo rpm --import \
        https://packages.elastic.co/GPG-KEY-elasticsearch
    echo "[elasticsearch-2.x]
name=Elasticsearch repository for 2.x packages
baseurl=http://packages.elastic.co/elasticsearch/2.x/centos
gpgcheck=1
gpgkey=http://packages.elastic.co/GPG-KEY-elasticsearch
enabled=1" | \
        sudo tee -a /etc/yum.repos.d/elasticsearch.repo
fi

# update list of available packages:
sudo yum update -y

# install Elasticsearch:
sudo yum install -y \
    elasticsearch \
    java

# allow network connections:
if ! sudo grep -q "network.host: ${INVENIO_ELASTICSEARCH_HOST}" \
    /etc/elasticsearch/elasticsearch.yml; then
    echo "network.host: ${INVENIO_ELASTICSEARCH_HOST}" | \
        sudo tee -a /etc/elasticsearch/elasticsearch.yml
fi

# open firewall ports:
if firewall-cmd --state | grep -q running; then
    sudo firewall-cmd --zone=public --add-port=9200/tcp --permanent
    sudo firewall-cmd --reload
fi

# enable Elasticsearch upon reboot:
sudo systemctl enable elasticsearch

# start Elasticsearch:
sudo systemctl start elasticsearch
```


Some packages require extra plugins to be installed.

```
$sudo /usr/share/elasticsearch/bin/plugin install -b mapper-attachments
```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
sudo apt-get -y autoremove && sudo apt-get -y clean
```

- on CentOS7:

```
sudo yum clean -y all
```

RabbitMQ

The RabbitMQ server (192.168.50.14) is used as a messaging middleware broker.

The RabbitMQ server can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-rabbitmq.sh
```

Let's see in detail what the RabbitMQ provisioning script does.

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
# update list of available packages:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y update

# install RabbitMQ server:
sudo DEBIAN_FRONTEND=noninteractive apt-get -y install \
    rabbitmq-server
```

- on CentOS 7:

```
# add EPEL external repository:
sudo yum install -y epel-release

# update list of available packages:
sudo yum update -y

# install Rabbitmq:
sudo yum install -y \
    rabbitmq-server

# open firewall ports:
if firewall-cmd --state | grep -q running; then
    sudo firewall-cmd --zone=public --add-port=5672/tcp --permanent
```

```
sudo firewall-cmd --reload
fi

# enable RabbitMQ upon reboot:
sudo systemctl enable rabbitmq-server

# start RabbitMQ:
sudo systemctl start rabbitmq-server
```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
sudo apt-get -y autoremove && sudo apt-get -y clean
```

- on CentOS7:

```
sudo yum clean -y all
```

Worker

The Celery worker node (192.168.50.15) is used to execute potentially long tasks in asynchronous manner.

The worker node can be set up in an automated unattended way by running the following script:

```
source .inveniorc
./scripts/provision-worker.sh
```

Let's see in detail what the worker provisioning script does.

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
echo "FIXME worker is a copy of web node"
```

- on CentOS 7:

```
echo "FIXME worker is a copy of web node"
```

Finally, let's clean after ourselves:

- on Ubuntu 14.04 LTS (Trusty Tahr):

```
sudo apt-get -y autoremove && sudo apt-get -y clean
```

- on CentOS7:

```
sudo yum clean -y all
```

Invenio

Now that all the prerequisites have been set up, we can proceed with the installation of the Invenio itself. The installation is happening on the web node (192.168.50.10).

We start by creating and configuring a new Invenio instance, continue by populating it with some example records, and finally we start the web application. This can be done in an automated unattended way by running the following scripts:

```
source .inveniorc
./scripts/create-instance.sh
./scripts/populate-instance.sh
./scripts/start-instance.sh
```

Note: If you want to install the very-bleeding-edge Invenio packages from GitHub, you can run the `create-instance.sh` script with the `--devel` argument:

```
./scripts/create-instance.sh --devel
```

Let's see in detail about every Invenio installation step.

Create instance

We start by creating a fresh new Python virtual environment that will hold our brand new Invenio v3.0 instance:

```
mkvirtualenv "${INVENIO_WEB_VENV}"
cdvirtualenv
```

We continue by installing Invenio v3.0 Integrated Library System flavour demo site from PyPI:

```
pip install invenio-app-ils[postgresql,elasticsearch2]
```

Let's briefly customise our instance with respect to the location of the database server, the Redis server, the Elasticsearch server, and all the other dependent services in our multi-server environment:

```
mkdir -p "var/instance/"
pip install "jinja2-cli>=0.6.0"
jinja2 "${scriptpathname}/instance.cfg" > "var/instance/${INVENIO_WEB_INSTANCE}.cfg"
```

In the instance folder, we run Npm to install any JavaScript libraries that Invenio needs:

```
${INVENIO_WEB_INSTANCE} npm
cdvirtualenv "var/instance/static"
CI=true npm install
```

We can now collect and build CSS/JS assets of our Invenio instance:

```
{INVENIO_WEB_INSTANCE} collect -v
{INVENIO_WEB_INSTANCE} assets build
```

Our first new Invenio instance is created and ready for loading some example records.

Populate instance

We proceed by creating a dedicated database that will hold persistent data of our installation, such as bibliographic records or user accounts. The database tables can be created as follows:

```
{INVENIO_WEB_INSTANCE} db init
{INVENIO_WEB_INSTANCE} db create
```

We continue by creating a user account:

```
{INVENIO_WEB_INSTANCE} users create \
    "{INVENIO_USER_EMAIL}" \
    --password "{INVENIO_USER_PASS}" \
    --active
```

We can now create the Elasticsearch indexes and initialise the indexing queue:

```
{INVENIO_WEB_INSTANCE} index init
sleep 20
{INVENIO_WEB_INSTANCE} index queue init
```

We proceed by populating our Invenio demo instance with some example demo MARCXML records:

```
{INVENIO_WEB_INSTANCE} demo init
```

Start instance

Let's now start the web application:

```
{INVENIO_WEB_INSTANCE} run -h 0.0.0.0 &
```

and the web server:

```
$sudo service nginx restart
```

We should now see our demo records on the web:

```
firefox http://{INVENIO_WEB_HOST}/records/1
```

and we can access them via REST API:

```
curl -i -H "Accept: application/json" \  
http://${INVENIO_WEB_HOST}/api/records/1
```

We are done! Our first Invenio v3.0 demo instance is fully up and running.

Configuration

Daemonizing applications

Monitoring

Error monitoring

Availability monitoring

Resource monitoring

High Availability

Load balancing

DNS load balancing

Distributed job queue

Geo-replication

Deploying with Fabric

Deploying with Docker

This part of the documentation will show you how to get started developing for Invenio.

First steps

Bootstrapping with cookiecutter

Invenio 3 has a tool to create a module from scratch, which allows you to have all the files needed for your new module. It is a template for Cookiecutter.

First, you need to install Cookiecutter, which is available on PyPI (the `-U` option will upgrade it if it is already installed):

```
pip install -U cookiecutter
```

Now we will create the files for the module. A module is basically a folder gathering all the files needed for its installation and execution. So, go where you want the directory to be created, and run the command:

```
cookiecutter https://github.com/inveniosoftware/cookiecutter-invenio-module.git
```

This will first clone the template from git to your current directory. Then, Cookiecutter will ask you questions about the module you want to create. Fill it this way (empty values take the default):

```
project_name [Invenio-FunGenerator]: Invenio-Unicorn
project_shortname [invenio-unicorn]:
package_name [invenio_unicorn]:
github_repo [inveniosoftware/invenio-unicorn]:
description [Invenio module that adds more fun to the platform.]:
author_name [CERN]: Nice Unicorn
author_email [info@inveniosoftware.org]: nice@unicorn.com
year [2017]:
copyright_holder [Nice Unicorn]:
copyright_by_intergovernmental [True]: False
```

```
superproject [Invenio]:
transifex_project [invenio-unicorn]:
extension_class [InvenioUnicorn]:
config_prefix [UNICORN]:
```

A folder `invenio-unicorn` has been created, you can go inside and have a look at all the generated files. If you want further information about the created files, you can read the *Invenio module layout* section.

Install, run and test

As soon as you have run the Cookiecutter, you have a valid module that can be installed. In this section, we are going to see how to install the module, run the tests, run the example application and build the documentation.

Before that, we need to **stop** any running Invenio instance.

Install the module

Install the module is very easy, you just need to go to the root directory of the module and run the command:

```
pip install -e .[all]
```

Some explanations about the command:

- the `-e` option is used for development. It means that if you change the files in the module, you won't have to reinstall it to see the changes. In a production environment, this option shouldn't be used.
- the `.` is in fact the path to your module. As we are in the root folder of the module, we can just say *here*, which is what the dot means
- **the `[all]` after the dot means we want to install all dependencies, which is common when developing. Depending on your**
 - the default (nothing after the dots) installs the minimum to make the module run
 - `[tests]` installs the requirements to test the module
 - `[docs]` installs the requirements to build the documentation
 - some modules have extra options

You can chain them: `[tests, docs]`.

Run the tests

In order to run the tests, you need to have a valid git repository. The following step needs to be run only once. Go in the root folder of the module:

```
git init
git add -A
check-manifest -u
```

What we have done:

- change the folder into a git repository, so it can record the changes made to the files
- add all the files to this repository
- update the file `MANIFEST.in`

Now, we are able to run the tests:

```
./run-tests.sh
```

Everything should pass as we didn't change any files yet.

Run the example application

The example application is a small app that presents the features of your module. By default, it simply prints a welcome page. To try it, go into the `examples` folder and run:

```
./app-setup.sh
./app-fixtures.sh
export FLASK_APP=app.py FLASK_DEBUG=1
flask run
```

You can now open a browser and go to the URL <http://localhost:5000/> you should be able to see a welcome page.

To clean the server, run the `./app-teardown.sh` script after killing the server.

Build the documentation

The documentation can be built with the `run-tests.sh` script, but you need the `tests` requirements, and run the tests. If you just want to build the documentation, you will only need the `docs` requirements (see the install section above) and run:

```
python setup.py build_sphinx
```

Before going further

Before going further in the tutorial, we need to push our repository to GitHub. The details about GitHub are explained in *Setting up your development environment*.

The first thing is to create a repo on GitHub, we will globally follow the GitHub documentation: <https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/>.

First, create an empty repository in your GitHub account. Be sure to not generate any `.gitignore` or `README` files, as our code already has them. If you don't have a GitHub account, you can skip this step, it is only necessary if you plan to publish your module on PyPI.

Now, go into the root directory of your module, and run

```
git remote add origin URL OF YOUR GITHUB REPO
```

Now, we can commit and push the generated files:

```
git commit -am "first commit"
git push --set-upstream origin master
```

Finally, we create a new branch to develop on it

```
git checkout -b dev
```

Develop a module

The goal of this tutorial is to add data to Invenio v3. We'll create a form that inserts the data in the database. Also we will touch different part of the development process such as:

- How to create a new view
- How to create a form
- How to add a utility function
- How to add new templates
- How to use Jinja2

Requirements

Before starting let's make sure we have `custom-data-module` installed on your environment. We need that to ensure that we have the data model.

1. Create the form

Ok, let's create a module that contains the forms of our project, we will use Flask-WTF.

in `invenio_unicorn/forms.py`

```
"""Forms module."""

from __future__ import absolute_import, print_function

from flask_wtf import FlaskForm
from wtforms import StringField, TextAreaField, validators

class RecordForm(FlaskForm):
    """Custom record form."""

    title = StringField(
        'Title', [validators.DataRequired()]
    )
    description = TextAreaField(
        'Description', [validators.DataRequired()]
    )
```

2. Create the views

in `invenio_unicorn/views.py` we'll create the endpoints for

- create: Form template
- success: Success template

The `views.py` registers all the views of our application

```
"""Invenio module that adds more fun to the platform."""

from __future__ import absolute_import, print_function
```

```

from flask import Blueprint, redirect, render_template, request, url_for
from flask_babel import gettext as _

from .forms import RecordForm
from .utils import create_record

blueprint = Blueprint(
    'invenio_unicorn',
    __name__,
    template_folder='templates',
    static_folder='static',
)

@blueprint.route("/")
def index():
    """Basic view."""
    return render_template(
        "invenio_unicorn/index.html",
        module_name=_('Invenio-Uncorn'))

@blueprint.route('/create', methods=['GET', 'POST'])
def create():
    """The create view."""
    form = RecordForm()
    # if the form is valid
    if form.validate_on_submit():
        # create the record
        create_record(
            dict(
                title=form.title.data,
                description=form.description.data
            )
        )
        # redirect to the success page
        return redirect(url_for('invenio_unicorn.success'))
    return render_template('invenio_unicorn/create.html', form=form)

@blueprint.route("/success")
def success():
    """The success view."""
    return render_template('invenio_unicorn/success.html')

```

3. Create the templates

And now, let's create the templates

in `invenio_unicorn/templates/invenio_unicorn/create.html` we override two blocks from the invenio `BASE_TEMPLATE` and those are:

- `javascript`
- `page_body`

In the javascript block we will right a small fetcher, to get the created records from the API, and in the page_body we will add the form and the placeholder for the records list.

```
{%- extends config.BASE_TEMPLATE %}

{% macro errors(field) %}
{% if field.errors %}
<span class="help-block">
  <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
  </ul>
{% endif %}
</span>
{% endmacro %}

{% block javascript %}
{{ super() }}
<script>
  $(document).ready(function() {
    $.get('/api/custom_records')
      .then(
        function(response) {
          $('#custom-records').html('');
          $.each(response.hits.hits, function(index, record) {
            $('#custom-records').append(
              '<li>' +
                '<h4><a href="/custom_records/' + record.metadata.custom_pid + "'>'
↪+ record.metadata.title + '</a></h4>' +
                '<p>' + record.metadata.description + '</p>' +
                '</li>'
              );
          });
        }, function() {
          $('#custom-records').html('');
        }
      );
  });
</script>
{% endblock javascript %}

{% block page_body %}
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <div class="alert alert-warning">
        <b>Heads up!</b> This example is for demo proposes only
      </div>
      <h2>Create record</h2>
    </div>
    <div class="col-md-offset-3 col-md-6 well">
      <form action="{{ url_for('invenio_unicorn.create') }}" method="POST">
        <div class="form-group" {{ 'has-error' if form.title.errors }}">
          <label for="title">{{ form.title.label }}</label>
          {{ form.title(class_="form-control")|safe }}
          {{ errors(form.title) }}
        </div>
        <div class="form-group" {{ 'has-error' if form.description.errors }}">
```

```

        <label for="description">{{ form.description.label }}</label>
        {{ form.description(class_="form-control")|safe }}
        {{ errors(form.description) }}
    </div>
    {{ form.csrf_token }}
    <button type="submit" class="btn btn-default">Submit</button>
</form>
</div>
</div>
</div>
<hr />
<div class="row">
    <div class="col-md-12">
        <h2>Records created</h2>
        <ol id="custom-records">
            <div class="text-center">
                Loading records...
            </div>
        </ol>
    </div>
</div>
</div>
</div>
{% endblock page_body %}

```

in invenio_unicorn/templates/invenio_unicorn/success.html

```

{%- extends config.BASE_TEMPLATE %}

{% block page_body %}
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <div class="alert alert-success">
                    <b>Success!</b>
                </div>
                <a href="{{ url_for('invenio_unicorn.create') }}" class="btn btn-warning">
↳ Create more</a>
                <hr />
                <center>
                    <iframe src="//giphy.com/embed/WZmgVLMt7mp44" width="480" height="480"
↳ frameborder="0" class="giphy-embed" allowFullScreen></iframe><p><a href="http://
↳ giphy.com/gifs/kawaii-colorful-unicorn-WZmgVLMt7mp44">via GIPHY</a></p>
                </center>
            </div>
        </div>
    </div>
{% endblock page_body %}

```

4. Create the record creation function

in invenio_unicorn/utils.py

On the `utils.py` module will create a helper function that creates a record.

```

"""Utils module."""

from __future__ import absolute_import, print_function

```

```
import uuid

from flask import current_app

from invenio_db import db
from invenio_indexer.api import RecordIndexer
from invenio_pidstore import current_pidstore
from invenio_records.api import Record

def create_record(data):
    """Create a record.

    :param dict data: The record data.
    """
    indexer = RecordIndexer()
    with db.session.begin_nested():
        # create uuid
        rec_uuid = uuid.uuid4()
        # add the schema
        host = current_app.config.get('JSONSCHEMAS_HOST')
        data["$schema"] = \
            current_app.extensions['invenio-jjsonschemas'].path_to_url(
                'custom_record/custom-record-v1.0.0.json')
        # create PID
        current_pidstore.minters['custid'](
            rec_uuid, data, pid_value='custom_pid_{}'.format(rec_uuid)
        )
        # create record
        created_record = Record.create(data, id_=rec_uuid)
        # index the record
        indexer.index(created_record)
    db.session.commit()
```

Demo time

Let's start our server again.

```
vagrant> invenio run -h 0.0.0.0
```

Then go to <http://192.168.50.10/create> and you will see the form we just created. There are two fields Title and Description.

Let's try the form, add something to the Title and click submit, you will see the validation errors on the form, fill in the Description and click submit. The form is now valid and it navigates you to the /success page.

Make your first commit

Now that we finished the development, we want to publish it on GitHub.

Push the code

To do so, we will first list our changes and add them to our local git repository:

```
git status
# shows all the files that have been modified
git add .
# adds all the modifications
```

Let's test our changes before we publish them. See *Run the tests* for more information.

```
./run-tests.sh
```

If it complains about the manifest, it is because we added new files, but we didn't register them into the `MANIFEST.in` file, so let's do so:

```
check-manifest -u
```

Once all the tests are passing, we can push our code. As we were developing on a branch created locally, we need to push the branch on GitHub:

```
git commit -am "basic development"
git push --set-upstream origin dev
```

Enable travis-cli and publish to GitHub

Configure travis-cli.org

1. Create an account

We need first to create an account on `travis-cli.org`.

Travis CI [Blog](#) [Status](#) [Help](#)

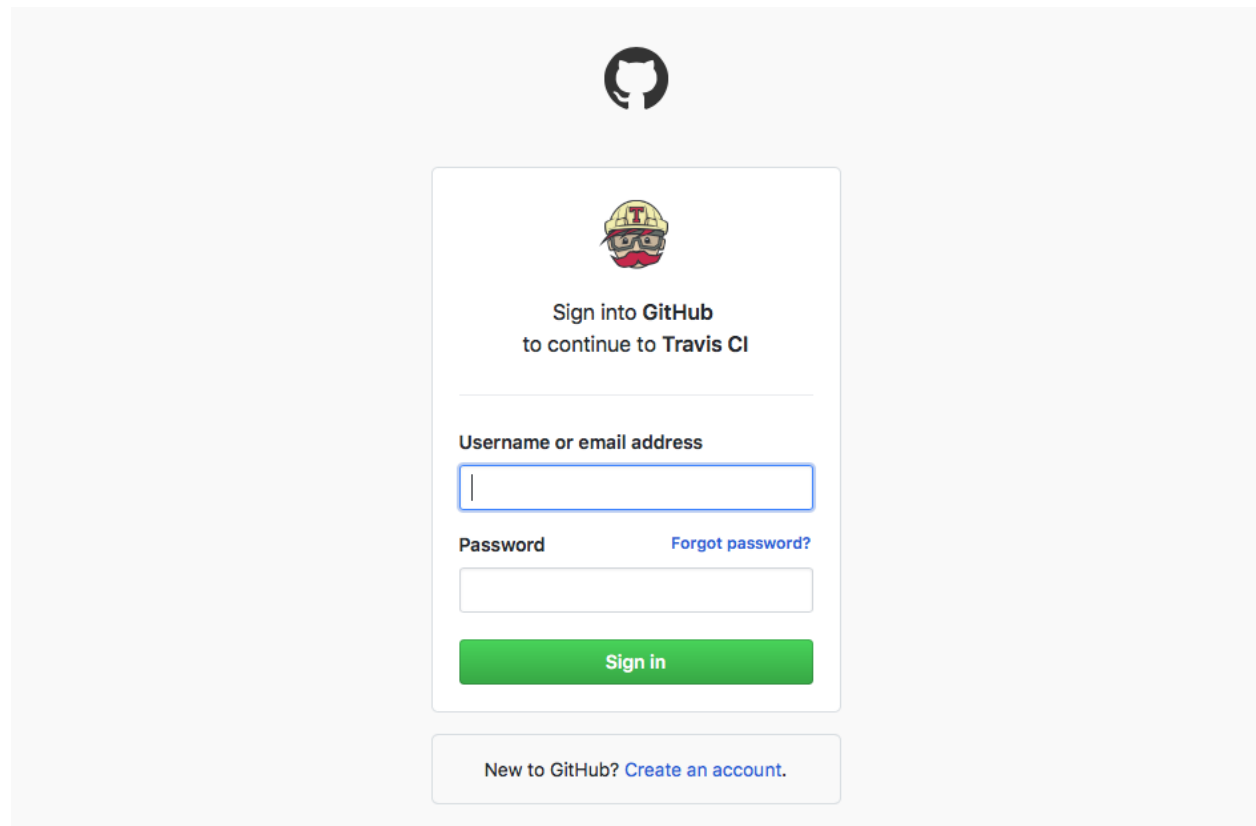
Sign in with GitHub 

Test and Deploy with Confidence

Easily sync your GitHub projects with Travis CI and you'll be testing your code in minutes!

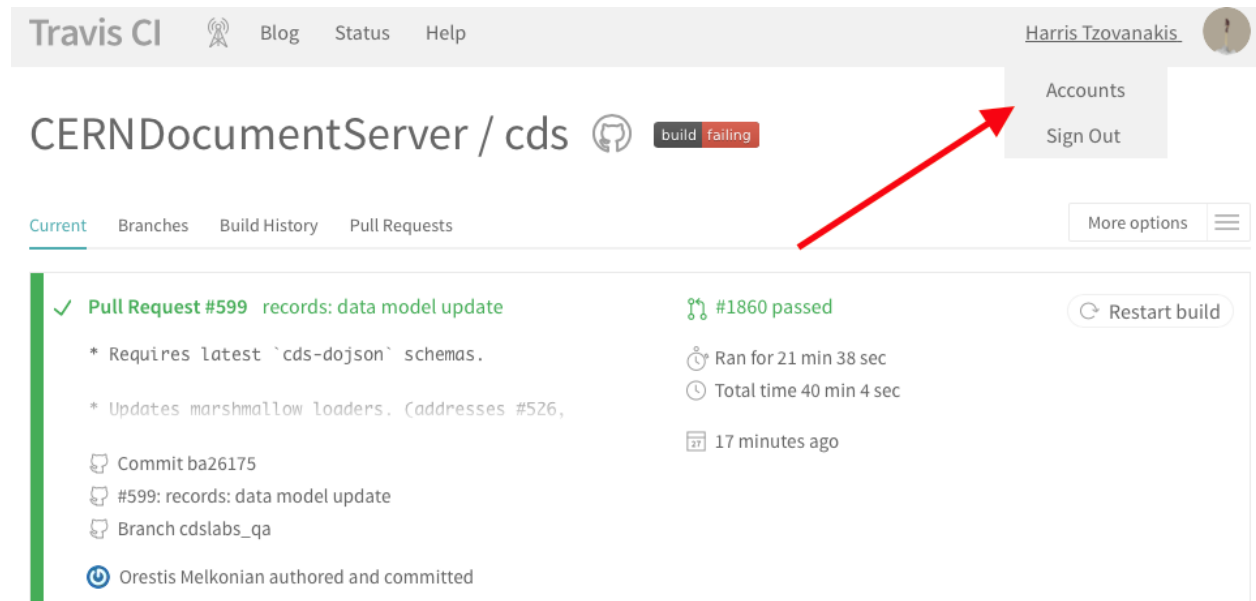


Add your github credentials to signup



2. Enable travis for your repo

Go to your account



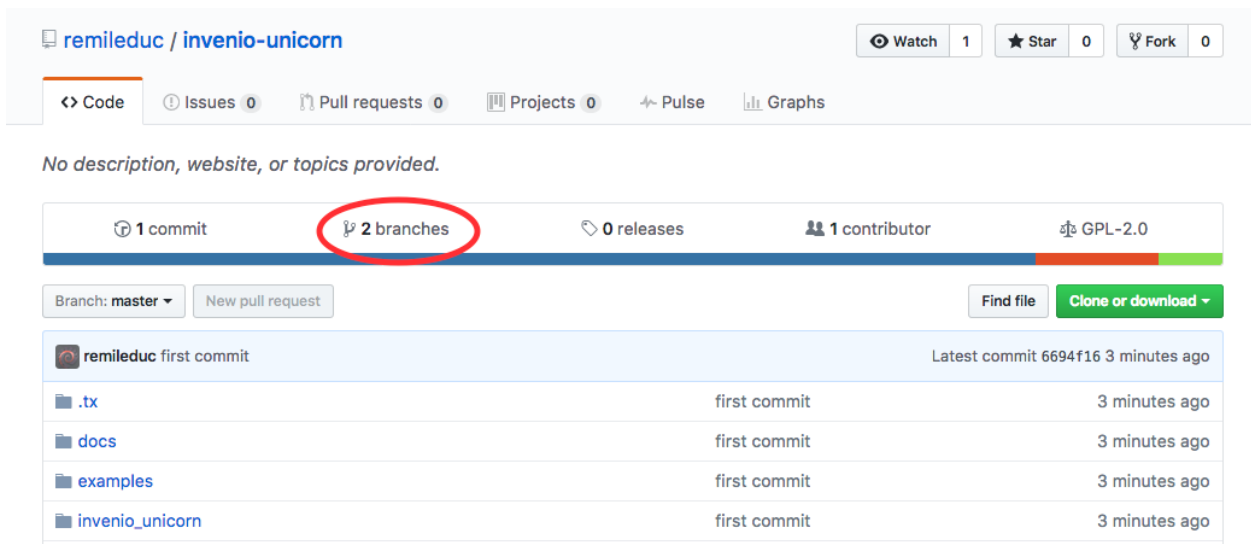
Click to enable the repo you have created



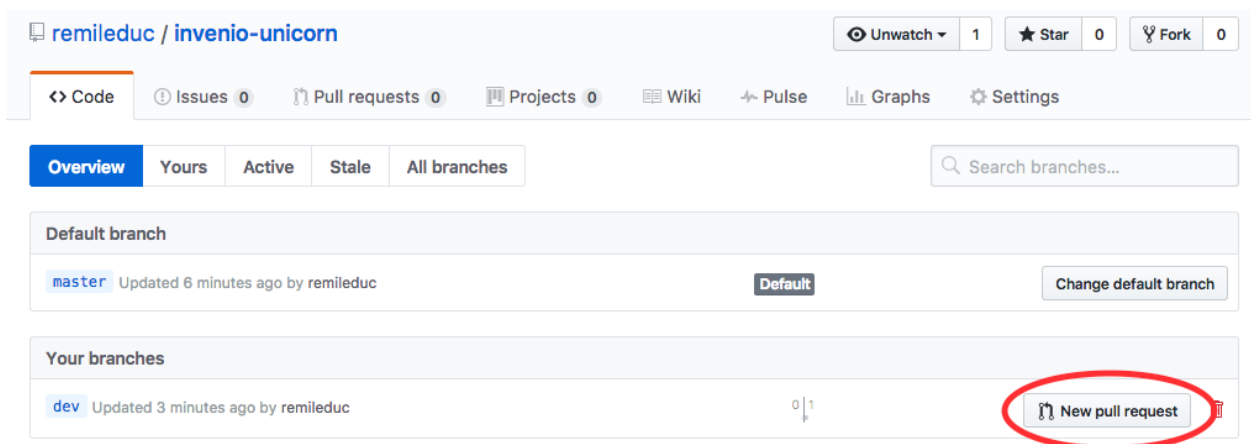
Done!

Create a Pull Request (PR)

We want that our changes get merged into the main branch (master) of the repository. So, let's go to the GitHub repository. From here, you can click on the *branch* button.



Then, click on *New pull request*



Now, you can check the differences that you will add to the main branch. Fill a description and create the pull request.

Setting up your development environment

Some rules should be applied when developing for Invenio. Here are some hints to set up your environment so these rules will be respected.

Git

Set up git

Git should be aware of your name and your e-mail address. To do so, run these commands:

```
git config --global user.name "Nice Unicorn"
git config --global user.email "nice@unicorn.com"
```

Editor

The most important rules are to make your editor print 4 spaces for indentation (instead of tabs) and limit the number of characters per line to 79.

Then, you can add automatic PEP8 checks if you want.

System architecture

Invenio v3.x

CAVEAT LECTOR

Invenio v3.0 alpha is a bleeding-edge developer preview version.

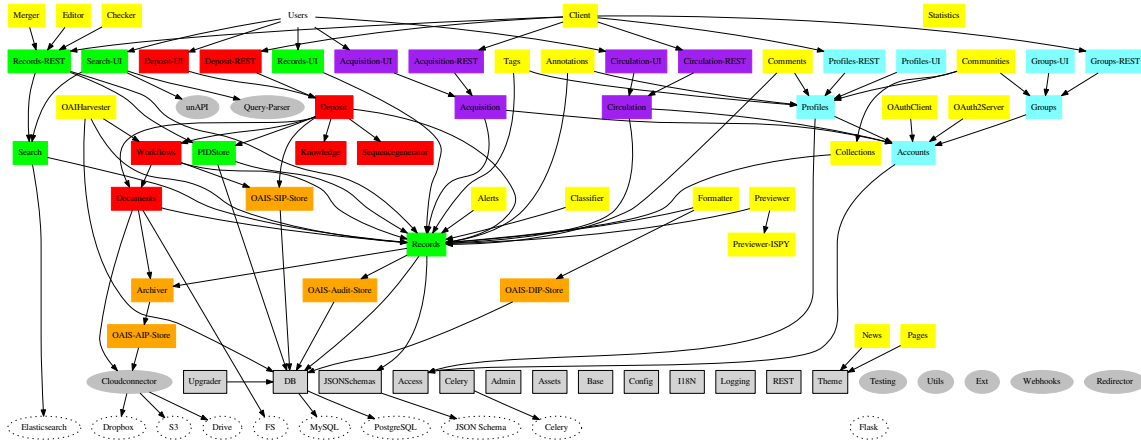
Invenio v3.0 build on top of [Flask](#) web development framework, using [Jinja2](#) template engine, [SQLAlchemy](#) Object Relational Mapper, [JSONSchema](#) data model, [PostgreSQL](#) database for persistence, and [Elasticsearch](#) for information retrieval.

Invenio's architecture is modular. The code base is split into more than 50 independent components that are [released independently on PyPI](#). This ensures strict separation of components that talk among themselves over API and permits rapid development of independent components by independent teams.

Invenio components, named *modules*, can be roughly split in three categories:

1. **base modules** provide interfaces to the Flask ecosystem, the Database, and other system tools and technologies that the Invenio ecosystem uses. Example: `Invenio-Celery` that talks to the Celery worker system.
2. **core feature modules** provide most common functionality that each digital library instance is likely interested in using. Example: `Invenio-Records` provide record object store.
3. **additional feature modules** offer additional functionality suitable for various particular use cases, such as the Integrated Library System, the Multimedia Store, or the Data Repository. Example: `Invenio-Circulation` offers circulation and holdings capabilities.

Here is a basic bird-eye overview of available Invenio components and their dependencies: (*work in progress*)



Application architecture

Invenio is at the core an application built on-top of the Flask web development framework, and fully understanding Invenio’s architectural design requires you to understand core concepts from Flask which will briefly be covered here.

The Flask application is exposed via different *application interfaces* depending on if the application is running in a webserver, CLI or job queue.

Invenio adds a powerful *application factory* on top of Flask, which takes care of dynamically assembling an Invenio application from the many individual modules that makes up Invenio, and which also allow you to easily extend Invenio with your own modules.

Core concepts

We will explain the core Flask concepts using simple Flask application:

```

from flask import Blueprint, Flask, request

# Blueprint
bp = Blueprint('bp', __name__)

@bp.route('/')
def my_user_agent():
    # Executing inside request context
    return request.headers['User-Agent']

# Extension
class MyExtension(object):
    def __init__(self, app=None):
        if app:
            self.init_app(app)

    def init_app(self, app):
        app.config.setdefault('MYCONF', True)

# Application

```

```
app = Flask(__name__)
ext = MyExtension(app)
app.register_blueprint(bp)
```

You can save above code in a file `app.py` and run the application:

```
$ pip install Flask
$ export FLASK_APP=app.py flask run
```

Application and blueprint

Invenio is a large application built up of many smaller individual modules. The way Flask allows you to build modular applications is via *blueprints*. In above example we have a small blueprint which just have one *view* (`my_user_agent`), which returns the browser's user agent sting.

This blueprint is *registered* on the *Flask application*. This allow you to possible reuse the blueprint in another Flask application.

Flask extensions

Like blueprints allow you to modularise your Flask application's views, then Flask extensions allow you to modularise non-view specific initialization of your application (e.g. providing database connectivity).

Flask extensions are just objects like the one in the example below, which has `init_app` method.

Application and request context

Code in a Flask application can be executed in two "states":

- *Application context*: when the application is e.g. being used via a CLI or running in a job queue (i.e. not handling requests).
- *Request context*: when the application is handling a request from a user.

In above example e.g. the code inside the view `my_user_agent` is executed during a request, and thus you can have access to the browser's user agent string. On the other hand, if you tried to access `request.headers` outside the view, the application would fail as no request is being processed.

The `request` object is a proxy object which points to the current request being processed. There is some magic happening behind the scenes in order to make this thread safe.

Interfaces: WSGI, CLI and Celery

Overall the Flask application is running via three different applications interfaces:

- **WSGI**: The frontend webservers interfaces with Flask via Flask's WSGI application.
- **CLI**: The command-line interface is made using Click and takes care of executing commands inside the Flask application.
- **Celery**: The disitributed job queue is made using Celery and takes care of executing jobs inside the Flask application.

Application assembly

In each of the above interfaces, a Flask application needs to be created. A common pattern for large Flask applications is to move the application creation into a factory function, named an **application factory**.

Invenio provides a powerful application factory for Flask which is capable of dynamically assembling an application. In order to illustrate the basics of what the Invenio application factory does, have a look at the following example:

```
from flask import Flask, Blueprint

# Module 1
bp1 = Blueprint(__name__, 'bp1')
@bp1.route('/')
def hello():
    return 'Hello'

# Module 2
bp2 = Blueprint(__name__, 'bp1')
@bp2.route('/')
def world():
    return 'World'

# Application factory
def create_app():
    app = Flask(__name__)
    app.register_blueprint(bp1)
    app.register_blueprint(bp2)
    return app
```

The example illustrates two blueprints, which are statically registered on the Flask application blueprint inside the application factory. It is essentially this part that the Invenio application factory takes care of for you. Invenio will automatically discover all your installed Invenio modules and register them on your application.

Assembly phases

The Invenio application factory assembles your application in five phases:

1. **Application creation:** Besides creating the Flask application object, this phase will also ensure your instance folder exists, as well as route Python warnings through the Flask application logger.
2. **Configuration loading:** In this phase your application will load your instance configuration. Your instance configuration is essentially all the configuration variables where you don't want to use the default values, e.g. the database host configuration.
3. **URL converter loading:** In this phase, the application will load any of your URL converts. This phase is usually only needed for some few specific cases.
4. **Flask extensions loading:** In this phase all the Invenio modules which provides Flask extensions will initialize the extension. Usually the extensions will provide default configuration values they need, unless the user already set them.
5. **Blueprints loading:** After all extensions have been loaded, the factory will end with registering all the blueprints provided by the Invenio modules on the application.

Understanding above application assembly phases, what they do, and how you can plug into them is essential for fully mastering Invenio development.

Note: No loading order within a phase

It's very important to know, that within each phase, there is **no order** in how the Invenio modules are loaded. Say, with in the Flask extensions loading phase, there's no way to specify that one extension has to be loaded before another extension.

You only have the order of the phases to work, so e.g. Flask extensions are loaded before any blueprints are loaded.

Module discovery

In each of the application assembly phases, the Invenio factory automatically discover your installed Invenio modules. The way this works, is via Python **entry points**. When you install the Python package for an Invenio module, the package describes via entry points which Flask extensions, blueprints etc. that this module provides. The section *Extending Invenio* describes in more detail how you use the entry points to extend Invenio.

WSGI: UI and REST

Each of the application interfaces (WSGI, CLI, Celery) may need slightly different Flask applications. The Invenio application factory is in charge of assembling these applications, which is done through the five assembly phases.

The WSGI application is however also split up into two Flask applications:

- **UI:** Flask application responsible for processing all user facing views.
- **REST:** Flask application responsible for processing all REST API requests.

The reason to split the frontend part of Invenio into two separate applications is partly

- to be able to run the REST API in one domain (`api.example.org`) and the UI app on another domain (`www.example.org`)
- because UI and REST API applications usually have vastly different requirements.

As an example, a 404 Not found HTTP error, usually needs to render a template in the UI application, but return a JSON response in the REST API application.

Implementation

The following Invenio modules are each responsible for implementing parts of above application architecture, and it is highly advisable to dig deeper into these modules if you want a better understanding of the Invenio application architecture:

- **Invenio-Base:** Implements the Invenio application factory.
- **Invenio-Config:** Implements the configuration loading phase.
- **Invenio-App:** Implements default applications for WSGI, CLI and Celery.

Extending Invenio

Invenio modules

Flask Extensions

Entry points

Quick word about entry-points: it is a mechanism widely used in Invenio 3.

Invenio is built on top of Flask, so it inherits its mechanisms: it is made of modules that you can add to get new features in your base installation.

In order to extend Invenio, modules use entry-points. There are a lot of available entry-points, like:

- *bundles* (to use CSS or JavaScript bundle)
- *models* (to store data in the database)
- ...

The complete list of entry points in Invenio can be seen running `invenio instance entrypoints`.

Depending on how your module extends Invenio, it will be registered on one or several entry points. A module can also add new entry points, thus the *bundles* entry point comes from `invenio_assets`, and its complete name is `invenio_assets.bundles`.

The entry-points used by your module are listed in the `setup.py` file.

Hooks

Signals

Invenio module layout

This page summarizes the standard structure and naming conventions of a module in Invenio v3.0. It serves as a reference point when developing a new module or enhancing an existing one.

A simple module may have the following folder structure:

```

invenio-foo/
  docs/
  examples/
  invenio_foo/
    templates/invenio_foo/
    __init__.py
    config.py
    ext.py
    version.py
    views.py
  tests/
  *.rst
  run-tests.sh
  setup.py

```

These files are described in the sections below.

Description of the files

*.rst files

All these files are used by people that wants to know more about your module (mainly developers).

- `README.rst` is used to describe your module. You can see the short description written in the Cookiecutter [here](#). You should update it with deeper details
- `AUTHORS.rst` should list all contributors to this module
- `CHANGES.rst` should be updated at every release and store the list of versions with the list of changes (changelog)
- `CONTRIBUTING.rst` presents the rules to contribute to your module
- `INSTALL.rst` describes how to install your module
- `RELEASE-NOTES.rst` should contain the most important notes about the current version

setup.py

First, there is the `setup.py` file, one of the most important: this file is executed when you install your module with `pip`. If you open it, you can see different parts.

On the top, the list of the requirements:

- for normal use
- for development
- for tests

Depending on your needs, you can install only part of the requirements, or everything (`pip install invenio-foo[all]`).

Then, in the `setup()` function, you have the description of your module with the values entered in the Cookiecutter. At the end, you can find the `entrypoints` section. For the moment, there is only the registration in the Invenio application, and the translations.

MANIFEST.in

This file lists all the interested files in the sub-folders. This file should be updated before the first commit. See the [Install, run and test](#) section.

run-tests.sh

This is used to run a list of tests locally, to be sure that your module works as you wish. It will generate the documentation, run `pytest` and do other checks.

This script should be run before every commits.

docs folder

This folder contains the settings to generate documentation for your module, along with files where you can write the documentation. When you run the `run-tests.sh` script, it will create the documentation in HTML files in a sub-folder.

examples folder

Here you can find a small example of how to use your module. You can test it, follow the steps described in the *Run the example application* section

tests folder

Here are described all the tests for your application, that will be run when you execute the `run-tests.sh` script. If all these tests pass, you can safely commit your work.

invenio_foo folder

This folder has the name of your module, in lower case with the dash changed into an underscore. Here is the code of your module. You can add any code files here, organized as you wish.

The files that already exist are kind of a standard, we are going through them in the following sections. The rule of thumbs here is that if you need multiple files for one action (for instance, 2 `views`: one for the API and a standard one), create a folder having the name of the file you want to split (here, a `views` folder with `ui.py` and `api.py` inside).

config.py

All your config variables should be declared in this file. Thus, if we look for how to customize your module, we just need to open this file.

ext.py

This is a specific file that you shouldn't touch except if you want to have advanced features. It contains a class that registers your module into the Invenio application, and load your default config variables.

version.py

Very basic file containing the version of your module.

views.py

Here you declare the views or end points you want to expose. By default, it creates a simple view on the root end point that fills a template.

templates

All your Jinja templates should be stored in this folder. A Jinja template is an HTML file that can be modified thanks to some parameters.

static

If your module needs JavaScript or CSS files, they should go in a folder called `static`. Also, if you want to group them in bundles, you should add a `bundles.py` file next to the `static` folder.

Module naming conventions

Invenio modules are standalone independent components that implement some functionality used by the rest of the Invenio ecosystem. The modules provide API to other modules and use API of other modules.

A module is usually called:

1. with plural noun, meaning “database (of things)”, for example `invenio-records`, `invenio-tags`, `invenio-annotations`,
2. with singular noun, meaning “worker (using things)”, for example `invenio-checker`, `invenio-editor`.

A module may have split its user interface and REST API interface, for example `invenio-records-ui` and `invenio-records-rest`, to clarify dependencies and offer easy customisation.

Bundles

Base bundle

Auth bundle

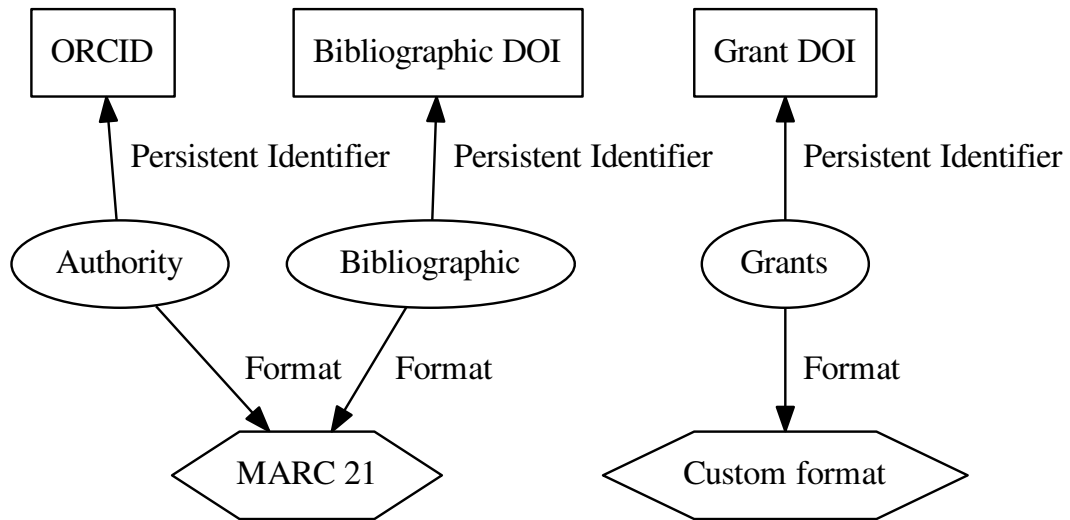
Metadata bundle

Future bundles

Creating a data model

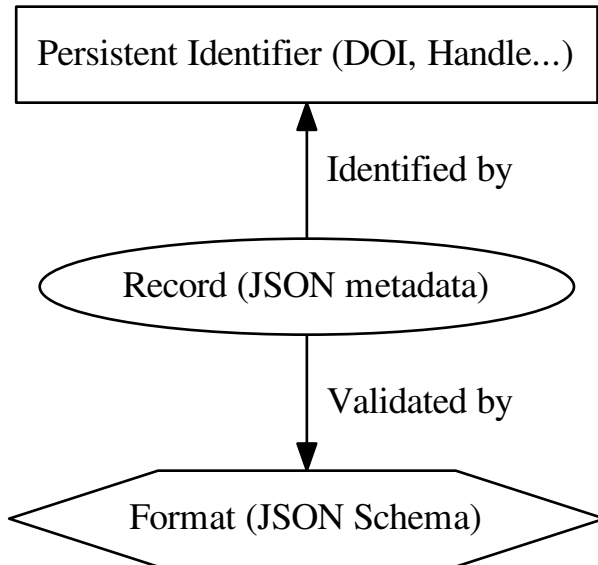
Use Cases

Invenio datamodel is designed for one pattern which is common to many different use cases: store and provide access to some **Records**, be it *Bibliographic* publications like research papers, a list of *Grants* or some *Authorities* like authors. Those resources are referenced by **Persistent Identifiers** and use a specific **format** for their **metadata**.



Records storage:

Invenio stores Record metadata as JSON, uses [JSON Schemas](#) for the formats and it can support any type of Persistent Identifier.



The JSON Schemas enable to define complex constraints which must be satisfied each time a record is created or modified. Invenio provides JSON Schemas for MARC 21 but custom JSON Schemas can also be used.

The rest of this documentation will show how to store and give access to documents having two metadata fields:

- title: the title of our document.
- description: the description of our documents.

Here is an example of such a record in the JSON format:

```

{
  "title": "CERN new accelerator",
  "description": "It now accelerates muffins."
}
  
```

Record Storage

JSON Schemas

The JSON Schema corresponding to our record could look like this:

```

{
  "title": "Custom record schema v1.0.0",
  "id": "http://localhost:5000/schemas/custom-record-v1.0.0.json",
}
  
```

```

"$schema": "http://json-schema.org/draft-04/schema#",
"type": "object",
"properties": {
  "title": {
    "type": "string",
    "description": "Record title."
  },
  "description": {
    "type": "string",
    "description": "Description for record."
  },
  "custom_pid": {
    "type": "string"
  },
  "$schema": {
    "type": "string"
  }
}

```

This JSON Schema defines the fields and their types. Other constraints can be added if needed.

Every record has a reference to the JSON Schema which validates it. In our example, the `$schema` field will be the URL pointing to the JSON Schema. The *invenio-jjsonschemas* module enables Invenio to serve JSON Schemas as static files.

We will explain *in the next part* why we added the “custom_pid” field.

External access to records:

Persistent Identifiers and URLs:

Different things can happen to published records. For example they can be:

- **deleted:** this happens when they contain invalid or illegal data. However we can’t just remove all information as users should be informed that the record existed at some point and was deleted.
- **merged:** duplicates are sometime created. It is then better to keep only one version. Users can be redirected from the deleted version to the one which was kept.

Invenio uses the concept of **Persistent Identifiers**, often abbreviated as **PID**. Those identifiers expose records to the outside world.

They are for example used in URLs. A typical User Interface url is of the form:

```
http://records/<PID>
```

Note that the *invenio-records-ui* module enables to customize the URL (ex: `http://authors/<PID>`), but it always contain the PID.

Persistent Identifiers can have different types and reflect Persistent Identifiers existing outside of Invenio such as DOI or ORCID. They can also be completely custom.

Many Invenio modules such as *invenio-records-ui* enable to have different configuration for each PID type. This for example enables to have one URL for authors and another for research papers.

Note: Records can have multiple Persistent Identifiers

One use case for multiple PIDs per records is systems which migrate from Invenio version 1 where records were referenced with incremental integers (ex: `http://records/1`). For backward compatibility reasons it is possible to keep internal PIDs which still use integers. The `invenio-pidstore` module provides everything needed for this use case. Our system might at the same time need to support DOI PIDs. It is then possible to create those PIDs without exposing them as an additional URL.

Note: PID minting

Every record's JSON contains a copy of its Persistent Identifier. We say that they are *minted* with the PID. The "custom_pid" field which we saw previously in the JSON Schema would contain this PID. This field name can be changed. It is advised to have it defined in the JSON Schema.

REST API:

Invenio enables access and modification of records via a REST API. This API is provided by the `invenio-records-rest` module, which uses Persistent Identifiers too.

A REST API URL will often look like:

```
http://api/records/<PID>
```

Note that just like `invenio-records-ui`, `invenio-records-rest` enables to customize the URLs for each PID type.

Serializers

The REST API can output records in any format as long as a **serializer** is defined. `invenio-marc21` provides serializers for MARC 21. Custom serializers can be easily added.

Here is a simple serializer example:

```
from flask import current_app

def plain_text_serializer(pid, record, code=200, headers=None, **kwargs):
    """Example of a custom serializer which just returns the record's title."""
    # create a response
    response = current_app.response_class()

    # set the returned data, which will just contain the title
    response.data = record['title']

    # set the return code in order to notify any error
    response.status_code = code

    # update headers
    response.headers['Content-Type'] = 'text/plain'
    if headers is not None:
        response.headers.extend(headers)
    return response
```

It is then possible to register this serializer for requestes of type `text/plain`. The result would look like this:

```
$ curl -H "Accept:text/plain" -XGET 'http://myinvenio.com/api/custom_records/custom_
↳pid_1'
CERN new accelerator
```

Serializers not only enable to output records in a specific format but also to remove fields, add fields or do any other transformation before showing the record to the outside world.

Search:

Users need to find records easily. Often this means to type a few words and get a list of results ordered by their relevance. Invenio uses Elasticsearch as its search engine. It needs to be configured in order to find the records as expected.

In this example we will focus on a very simple use case: how to search records containing english text in its metadata. This means that if our record contains “muffins” it should also be found when the user queries with the word “muffin” (without ‘s’).

We will provide an **Elasticsearch mapping** file which will define every field and specify that it should be *analyzed* as “english”.

```
{
  "mappings": {
    "custom-record-v1.0.0": {
      "_all": {
        "analyzer": "english"
      },
      "properties": {
        "title": {
          "type": "string",
          "analyzer": "english"
        },
        "description": {
          "type": "string",
          "analyzer": "english"
        },
        "custom_pid": {
          "type": "string",
          "index": "not_analyzed"
        },
        "$schema": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

If you want to know more about Elasticsearch mapping you can see its documentation.

Linking records:

Invenio provides tools to link records one to another.

We can extend our example by adding a “references” field which will contain a list of references to other records.

When creating a record the user would give this as input:

```
{
  "title": "CERN new accelerator",
  "description": "It now accelerates muffins.",
  "references": [
    {"$ref": "http://myinvenio.com/custom_records/custom_pid_1#/title" },
    {"$ref": "http://myinvenio.com/custom_records/custom_pid_42#/title" }
  ]
}
```

The pattern `{"$ref": "http://myinvenio.com/records/1#/title" }` is called a JSON reference. It enables to have a reference to another JSON object, or a field in it, with a URL just like `$schema`.

The corresponding JSON Schema would be:

```
{
  "title": "Custom record schema v1.0.0",
  "id": "http://localhost:5000/schemas/custom-record-v1.0.0.json",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "title": {
      "type": "string",
      "description": "Record title."
    },
    "description": {
      "type": "string",
      "description": "Description for record."
    },
    "references": {
      "type": "array",
      "items": {
        "type": "object"
      }
    },
    "custom_pid": {
      "type": "string"
    },
    "$schema": {
      "type": "string"
    }
  }
}
```

Invenio provide tools to dereference those JSON references and replace them with the referenced value. The output would then look like this:

```
$ curl -XGET 'http://myinvenio.com/api/custom_records/custom_pid_1'
{
  "created": "2017-03-16T14:53:42.126710+00:00",
  "links": {
    "self": "http://192.168.50.10/api/custom_records/custom_pid_1"
  },
  "metadata": {
    "$schema": "http://myinvenio.com/schema/custom_record/custom-record-v1.0.0.json",
    "custom_pid": "custom_pid_1",
    "title": "CERN new accelerator",
    "description": "It now accelerates muffins."
    "references": [
```



```

    "This is the title of record custom_pid_1",
    "This is the title of record custom_pid_42",
  ]
},
"updated": "2017-03-16T14:53:42.126725+00:00"
}

```

The dereferencing is done by the serializer. The database still contain the original JSON references.

This dereferencing is also done before the record is indexed in Elasticsearch. Thus the mapping would define the “references” field as a list of string (titles are of type string):

```

{
  "mappings": {
    "custom-record-v1.0.0": {
      "_all": {
        "analyzer": "english"
      },
      "properties": {
        "title": {
          "type": "string",
          "analyzer": "english"
        },
        "description": {
          "type": "string",
          "analyzer": "english"
        },
        "references": {
          "type": "string"
        },
        "custom_pid": {
          "type": "string",
          "index": "not_analyzed"
        },
        "$schema": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}

```

Warning: The records containing the references need to be reindexed if the referenced records change.

Docker

This page describes how to set up Docker containers for development purposes.

Setup

Install Docker and Docker Compose. Now run:

```
docker-compose -f docker-compose-dev.yml build
docker-compose -f docker-compose-dev.yml up
```

This builds and runs the docker containers. You can now connect to `localhost:28080` to see your Invenio installation. The `admin` user does not have any password.

Caution: This will mirror the current source code directory into the Docker container, so make sure to delete all `*.pyc`-files before. They might not be compatible with the Python version and libraries of Docker image.

Note: If you are using `boot2docker` you need to set up port forwarding by running the following command in a new terminal:

```
boot2docker ssh -vnNT \
  -Llocalhost:29200:localhost:29200 \
  -Llocalhost:28080:localhost:28080 \
  -Llocalhost:26379:localhost:26379 \
  -Llocalhost:25673:localhost:25673 \
  -Llocalhost:25672:localhost:25672 \
  -Llocalhost:23306:localhost:23306
```

You have to run this after Invenio booted up. Do **not** stop it while you are working with Invenio. Otherwise the port forwarding gets stopped. You have to restart the forwarding when you restart the Docker containers. The process can be stopped using `CTRL-C`.

Should you require a fresh installation and therefore wipe all your instance data, run:

```
docker-compose -f docker-compose-dev.yml rm -f
```

Interactive sessions

For interactive sessions inside the container simply attach a shell to the running instance:

```
docker exec -it invenio_web_1 bash
```

Note: Some tools (mostly ncurses-based ones) require the ability to detect the used terminal. To enable this, set the `TERM` environment variable to the correct value as a first command inside the container, e.g. by running:

```
export TERM=xterm
```

Note: Do not forget to run `bibsched` to schedule the processing of uploaded records. You can put it into automatic mode if you like.

Debugging

The `docker-compose-dev.yml` enables `Werkzeug`, a debugger that automatically kicks in whenever an error occurs. Stacktraces and debugger terminal are available via web interface.

You can also insert a tracepoint into the code to start an interactive debugger session:

```
import ipdb; ipdb.set_tracepoint()
```

Furthermore you can debug MySQL at `localhost:23306`, Elasticsearch at `localhost:29200`, RabbitMQ via `localhost:25672` (webinterface at `localhost:25673`) and Redis at `localhost:26379`. You might want to use `flower` for celery debugging and analysis as well. Just run the following command to open the webinterface at port 5555:

```
celery flower --broker=amqp://guest:guest@localhost:25672//
```

Should you require additional information about the behaviour of the different containers as well as the contained processes and their interaction with the system and other processes, the usage of classical Linux tools like `Wireshark` and `sysdig` might be helpful.

Code changes and live reloading

Note: This section does not apply to OS X, Windows and boot2docker as these systems are not properly supported by the used watchdog mechanism. When you are using one of these setups, you have to restart the Docker containers to reload the code and templates.

As long as you do not add new requirements (python and npm) and only change files inside the `invenio` package, it is not required to rebuild the docker images. Code changes are mirrored to the containers. If Flask supports it, on your system it will automatically reload the application when changes are detected. This sometimes might lead to timeouts in your browser session. Do not worry about this, but be aware to only save files when you are ready for reloading.

As of this writing changing template files do not lead to application reloading and do not purge caches. As a workaround you can simple alter one of the python files, e.g. by using `touch`.

Note: Changing the Python source files will invalidate stored bytecode files. For security reasons, these bytecode files can only be recreated by the root user. This can be done via:

```
docker exec -it -u root invenio_web_1 python -O -m compileall .
```

Building documentation

You can also use the Docker container to build the documentation. This can be done by attaching to running container:

```
docker exec -it invenio_web_1 sphinx-build -nW docs docs/_build/html
```

Note: This needs do be done in a running or initialized container because it requires that Invenio is set up correctly. Otherwise, the script will break because of missing access rights.

Running tests

You can also run tests using the Docker containers. Wait until the containers finished setup and the webservice is running. Then use:

```
docker exec -it invenio_web_1 python setup.py test
```

Note: Running the test requires the deactivation of redirection debugging. You can achieve this by setting the configuration variable `DEBUG_TB_INTERCEPT_REDIRECTS = False`. (Done for you by default if you use `docker-compose`.)

Overlays

You might want to use build distribute overlays using Docker. Instead of creating an entire new image and rewrite everything from scratch, you can the Invenio Docker image. Start by building the image from a branch or release of your choice:

```
cd src/invenio
docker build -t invenio:2.0 .
```

Now go to your overlay and create a Dockerfile that suits your needs, e.g:

```
# based on the right Invenio base image
FROM invenio:2.0

# get root rights again
USER root

# optional:
# add new packages
# (update apt caches, because it was cleaned from the base image)
# RUN apt-get update && \
#     apt-get -qy install whatever_you_need

# optional:
# add new packages from pip
# RUN pip install what_suits_you

# optional:
# add new packages from npm
# RUN npm update && \
#     npm install fun

# optional:
# make even more modifications

# add content
ADD . /code-overlay
WORKDIR /code-overlay

# fix requirements.txt and install additional dependencies
RUN sed -i '/inveniosoftware\/invenio[@#]/d' requirements.txt && \
    pip install -r requirements.txt --exists-action i
```

```
# build
RUN python setup.py compile_catalog

# optional:
# do some cleanup

# step back
# in general code should not be writeable, especially because we are using
# `pip install -e`
RUN mkdir -p /code-overlay/src && \
  chown -R invenio:invenio /code-overlay && \
  chown -R root:root /code-overlay/invenio_demosite && \
  chown -R root:root /code-overlay/scripts && \
  chown -R root:root /code-overlay/setup.* && \
  chown -R root:root /code-overlay/src

# finally step back again
USER invenio
```

Notice that this Dockerfile must be located in the directory of your overlay. For a full working example, please see [invenio-demosite](#). Here is how to build the demo site:

```
cd ~/private/src/invenio
git checkout maint-2.0
docker build -t invenio:2.0 .
cd ~/private/src/invenio-demosite
git checkout maint-2.0
docker-compose -f docker-compose-dev.yml build
docker-compose -f docker-compose-dev.yml up
```

After all the daemons are up and running, you can populate the demo site with demo records:

```
docker exec -i -t -u invenio inveniodemosite_web_1 \
  inveniomange demosite populate \
  --packages=invenio_demosite.base --yes-i-know
```

Done. Your Invenio overlay installation is now up and running.

W6. Rebasing against latest git/master

At this step the new-feature-b code is working both for Atlantis and for CDS contexts. You should now check the official repo for any updates to catch any changes that may have been committed to origin/master in the meantime.

```
$ git checkout master
$ git pull
```

You can then **rebase** your new-feature-b branch against recent master.

```
$ git checkout new-feature-b
$ git rebase master
```

In case of conflicts during the rebase, say in file foo.py, you should resolve them.

```
$ vim foo.py
$ git add foo.py
$ git rebase --continue
```

or you can stop the rebase for good.

```
$ git rebase --abort
```

You may prefer rebasing of your local commits rather than merging, so that the project log looks nice. (No ugly empty merge commits, no unnecessary temporary versions.)

While rebasing, you may want to squash your commits together, to keep the git repo history clean. See section R4 below for more details.

You should test your code once more to verify that it was not broken by the updates.

Learning Python

Anti-patterns

Learn to recognise and avoid Python anti-patterns.

3. Contributing code?

- (a) Read [The Zen of Python](#).
- (b) Read [Python Anti-Patterns](#).
- (c) Code with style. Plug [pycodestyle \(PEP-8\)](#), [pydocstyle \(PEP-257\)](#), [Flake8](#), [ISort](#) tools into your editor.

Entry points

We use Python `entry_points` as a way to provide extensions for packages.

Notes on getting involved, contributing, legal information and release notes are here for the interested.

Getting help

Didn't find a solution to your problem the Invenio documentation? Here's how you can get in touch with other users and developers:

Forum/Knowledge base

- <https://github.com/inveniosoftware/troubleshooting>

Ask questions or browse answers to existing questions.

Chatroom

- <https://gitter.im/inveniosoftware/invenio>

Probably the fastest way to get a reply is to join our chatroom. Here most developers and maintainers of Invenio hangout during their regular working hours.

GitHub

- <https://github.com/inveniosoftware>

If you have feature requests or want to report potential bug, you can do it by opening an issue in one of the individual Invenio module repositories. In each repository there is a MAINTAINERS file in the root, which lists the who is maintaining the module.

Communication channels

Chatrooms

Most day-to-day communication is happening in our chatrooms:

- [Public chatroom](#) (for everyone)
- [Developer chatroom](#) (for members of inveniosoftware GitHub organisation).

If you want to join the developer chatroom, just ask for an invite on the public chatroom.

GitHub

Most of the developer communication is happening on GitHub. You are strongly encouraged to join <https://github.com/orgs/inveniosoftware/teams/developers> team and watch notifications from various <https://github.com/inveniosoftware/> organisation repositories of interest.

Invenio Developer Forum

Monday afternoons at 16:00 CET/CEST time we meet physically at CERN and virtually over videoconference to discuss interesting development topics.

You can join our Monday forums from anywhere via videoconference. Here the steps:

- View the [agenda](#) (ical feed).
- Install the videoconferencing client [Vidyo](#).
- Join our [virtual room](#).

Invenio User Group Workshop

We meet among Invenio users and developers from around the world at a yearly Invenio User Group Workshop. The workshop consists of a series of presentations, tutorials, practical exercises, and discussions on topics related to Invenio digital library management and development. We exchange knowledge and experiences and drive forward the forthcoming developments of the Invenio platform.

See list of [upcoming](#) and [past](#) workshops.

Project website

Our project website, <http://inveniosoftware.org>, is used to show case Invenio.

Todo

Make better description of project website. Add high level road map etc to project website.

Email

You can get in touch with the Invenio management team on info@inveniosoftware.org.

In particular use above email address to report security related issues privately to us, so we can distribute a security patch before potential attackers look at the issue.

Twitter

The official Twitter account for Invenio software is used mostly for announcing new releases and talks at conferences:

- <https://twitter.com/inveniosoftware>

Mailing lists

The mailing lists are currently not very active and are primarily listed here for historical reasons.

- project-invenio-announce@cern.ch: Read-only moderated mailing list to announce new Invenio releases and other major news concerning the project. [subscribe to announce](#), [archive](#)
- project-invenio-general@cern.ch: Originally used for discussion among users and administrators of Invenio instances. The mailing list has mostly been replaced by our public chatroom. [subscribe to general](#), [new general archive](#), [old general archive](#)
- project-invenio-devel@cern.ch: Originally used for discussion among Invenio developers. The mailing list has mostly been replaced by our developer chatroom. [subscribe to devel](#), [new devel archive](#), [old devel archive](#)

Note that all the mailing lists are also archived (as of the 20th of July, 2011) on [The Mail Archive](#).

Todo

Setup an architecture team meeting between core maintainers.

Contribution guide

Interested in contributing to the Invenio project? There are lots of ways to help.

Code of conduct

Overall, we're **open, considerate, respectful and good to each other**. We contribute to this community not because we have to, but because we want to. If we remember that, our *Code of Conduct* will come naturally.

Get in touch

See [Getting help](#) and [Communication channels](#). Don't hesitate to get in touch with the Invenio maintainers (listed in MAINTAINERS file). The maintainers can help you kick start your contribution.

Types of contributions

Report bugs

- **Found a bug? Want a new feature?** Open a GitHub issue on the applicable repository and get the conversation started (do search if the issue has already been reported). Not sure exactly where, how, or what to do? See *Getting help*.
- **Found a security issue?** Alert us privately at info@inveniosoftware.org, this will allow us to distribute a security patch before potential attackers look at the issue.

Translate

- **Missing your favourite language?** Translate Invenio on [Transifex](#)
- **Missing context for a text string?** Add context notes to translation strings or report the issue as a bug (see above).
- **Need help getting started?** See our *Translation guide*.

Write documentation

- **Found a typo?** You can edit the file and submit a pull request directly on GitHub.
- **Debugged something for hours?** Spare others time by writing up a short troubleshooting piece on <https://github.com/inveniosoftware/troubleshooting/>.
- **Wished you knew earlier what you know now?** Help write both non-technical and technical topical guides.

Write code

- **Need help getting started?** See our *Developer's Guide*.
- **Need help setting up your editor?** See our *Setting up your development environment* guide which helps your automate the tedious tasks.
- **Want to refactor APIs?** Get in touch with the maintainers and get the conversation started.
- **Troubles getting green light on Travis?** Be sure to check our *Style guide TODO* and the *Setting up your development environment*. It will make your contributor life easier.
- **Bootstrapping a new awesome module?** Use our [Invenio cookiecutter template](#).

Style guide (TL;DR)

Travis CI is our style police officer who will check your pull request against most of our *Style guide TODO*, so do make sure you get a green light from him.

ProTip: Make sure your editor is setup to do checking, linting, static analysis etc. so you don't have to think. Need help setting up your editor? See *Setting up your development environment*.

Commit messages

Commit message is first and foremost about the content. You are communicating with fellow developers, so be clear and brief.

(Inspired by [How to Write a Git Commit Message](#))

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Indicate the component follow by a short description
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how, using bullet points

ProTip: Really! Spend some time to ensure your editor is top tuned. It will pay off many-fold in the long run. See *Setting up your development environment*.

For example:

```
component: summarize changes in 50 char or less

* More detailed explanatory text, if necessary. Formatted using
  bullet points, preferably `*`. Wrapped to 72 characters.

* Explain the problem that this commit is solving. Focus on why you
  are making this change as opposed to how (the code explains that).
  Are there side effects or other unintuitive consequences of this
  change? Here's the place to explain them.

* The blank line separating the summary from the body is critical
  (unless you omit the body entirely); various tools like `log`,
  `shortlog` and `rebase` can get confused if you run the two
  together.

* Use words like "Adds", "Fixes" or "Breaks" in the listed bullets to help
  others understand what you did.

* If your commit closes or addresses an issue, you can mention
  it in any of the bullets after the dot. (closes #XXX) (addresses
  #YYY)

Co-authored-by: John Doe <john.doe@example.com>
```

Pull requests

Need help making your first pull request? Check out the GitHub guide [Forking Projects](#).

When making your pull request, please keep the following in mind:

- Create logically separate commits for logically separate things.
- Include tests and don't decrease test coverage.
- Do write documentation. We all love well-documented frameworks, right?

- Run tests locally using `run-tests.sh` script.
- Make sure you have the rights if you include third-party code (and do credit the original creator).
- Green light on all GitHub status checks is required in order to merge your PR.

Work in progress (WIP)

Do publish your code as pull request sooner than later. Just prefix the pull request title with `WIP` (=work in progress) if it is not quite ready.

Allow edits from maintainers

To speed up the integration process, it helps if on GitHub you [allow maintainers to edit your pull request](#) so they can fix small issues autonomously.

Style guide TODO

Python

We follow [PEP-8](#) and [PEP-257](#) and sort imports via `isort`. Please plug corresponding linters such as `flake8` to your editor.

Commit message

Todo

Needs to be combined from existing parts and write the missing parts.

Indicate the component follow by a short description

We know space is precious and 50 characters is not much for the subject line, but maintainers and other people in general will appreciate if you can narrow the focus of your commit in the subject.

Normal components match to the python modules inside the repository, i.e. `api`, `views`, or `config`. There are other components which correspond to a wider scope like `tests` or `docs`. And finally there is third category which doesn't correspond to any file or folder in particular:

- *installation*: use it when modifying things like requirements files or `setup.py`.
- *release*: only to be used by maintainers.

If one commit modifies more than one file, i.e. `api.py` and `views.py`, common sense should be applied, what represents better the changes the commit makes? Remember you can always ask for the modules maintainer's opinion.

Use the body to explain what and why vs. how using bullet points

Take a look at the full diff and just think how much time you would be saving fellow and future committers by taking the time to provide this context here and now. If you don't, it would probably be lost forever.

In most cases, you can leave out details about how a change has been made.

In most cases, you can leave out details about how a change has been made. Code is generally self-explanatory in this regard (and if the code is so complex that it needs to be explained in prose, that's what source comments are for). Just focus on making clear the reasons why you made the change in the first place—the way things worked before the change (and what was wrong with that), the way they work now, and why you decided to solve it the way you did. Using bullet points will help you be precise and direct to the point.

If you find your self writing a rather long commit message, maybe it's time to step back and consider if you are doing too many changes in just one commit and whether or not it's worth splitting it in smaller peaces.

And remember, the future maintainer that thanks you for writing good commit messages may be yourself!

Submitting a pull request

All proposed changes to any of the Invenio modules are made as GitHub pull requests, if this is the first time you are making a contribution using GitHub, please check [this](#).

Once you are ready to make your pull request, please keep in mind the following:

- Before creating your pull request run the `run-tests.sh` script, this will help you discover possible side effects of your code and ensure it follows [Invenio's style guidelines](#), check [Development Environment](#) for more information on how you can run this script.
- Every pull request should include tests to check the expected behavior of your changes and must not decrease test coverage. If it fixes a bug it should include a test which proves the incorrect behavior.
- Documenting is part of the development process, no pull request will be accepted if there is missing documentation.
- No pull request will be merged until all automatic checks are green and at least one maintainer approves it.

Maintainers

The Invenio project follows a similar maintainer phylosofy as [docker](#). If you want to know more about it or take part, you can read our [Maintainer's guide](#).

FAQ

Translation guide

Invenio has been translated to more than 25 languages.

Transifex

All Invenio internationalisation and localisation efforts are happening on the [Transifex](#) collaborative localisation platform.

You can start by exploring [Invenio project dashboard](#) on Transifex.

Invenio project consists of many resource files which can be categorised as follows:

- `invenio-maint10-messages` contains phrases for Invenio 1.0 legacy release series
- `invenio-maint11-messages` contains phrases for Invenio 1.1 legacy release series
- `invenio-maint12-messages` contains phrases for Invenio 1.2 legacy release series
- `invenio-xxx-messages` contains phrases for Invenio 3.0 `invenio-xxx` module

We follow the usual Transifex localisation workflow that is [extensively documented](#).

Translators

All contributions with translating phrases or with reviewing translations are very appreciated!

Please read [Getting started as a translator](#) and join Invenio project on Transifex.

Developers

Please see dedicated [invenio-i18n](#) documentation.

Maintainer's guide TODO

Todo

Fill with new proposal

Overview

What is a maintainer?

(Clearly inspired by the [Docker](#) project)

There are different types of maintainers in the Invenio project, with different responsibilities, but all of them have this **rights and responsibilities** in common:

1. Have merge rights in the repositories they are maintainers of, but no pull request can be merged until all checks have passed and at least one maintainer signs off. (If one maintainer is making a pull request another maintainer should sign off, this will prevent self merges)
2. Have the final word about architectural or API significant changes, ensuring always that the [Invenio deprecation policies](#orgheadline1) are followed.
3. Will review pull requests within a reasonable time range, offering constructive and objective comments in a respectful manner.
4. Will participate in feature development and bug fixing, proactively verifying that the nightly builds are successful.
5. Will prepare releases following the Invenio standards.
6. Will answer questions and help users in the [Invenio Gitter chat room](#).
- 7.

Core maintainers

The core maintainers are the architectural team who shapes and drives the Invenio project and, as a consequence, they are the ultimate responsible of the success of it.

They are ghostbusters of the project: when there's a problem others can't solve, they show up and fix it with bizarre devices and weaponry. Some maintainers work on the project Invenio full-time, although this is not a requirement.

For each release (including minor releases), a "release captain" is assigned by the core maintainers from the pool of module maintainers. Rotation is encouraged across all maintainers, to ensure the release process is clear and up-to-date. The release can't be done without a core maintainer approvals.

Any of the core maintainers can propose new module maintainers at any time, see *Becoming a maintainer* for more information.

Module maintainers

Module maintainers are exceptionally knowledgeable about some but not necessarily all areas of the Invenio project, and are often selected due to specific domain knowledge that complements the project (but a willingness to continually contribute to the project is most important!).

The duties of a module maintainer are very similar to those of a core maintainer, but they are limited to modules of the Invenio project where the module maintainer is knowledgeable. Those who have write access to the Invenio-X repository**. All maintainers can review pull requests and add LGTM labels as appropriate, in fact everyone is encouraged to do so!

Becoming a maintainer

Don't forget: being a maintainer is a time investment. Make sure you will have time to make yourself available. You don't have to be a maintainer to make a difference on the project!

Usually to become a module maintainer, one of the core maintainers makes a pull request in the opensource project to propose the new maintainer. This pull request needs the approval from other core maintainer and from at least one of the module maintainers (if the module has any).

To become a core maintainer the process is slightly different. Before being a core maintainer candidate one needs to make sustained contributions to the project over a period of time (usually more than 3 months), show willingness to help Invenio users on GitHub and in the [Invenio Gitter chat room](#), and, overall, a friendly attitude. Once the above is out of question, the core maintainer who sponsors the future one will present him to the team and, if he gets the confidence of all of the current core maintainers, his sponsor will make a pull request to the opensource repository adding his name to the list.

Stepping down as maintainer

Stepping down as a maintainer is done also via pull request to the opensource GitHub repository. It can be the maintainer himself who makes the pull request and, as before, this type of pull requests need to be approved by one core maintainer and at least one of the module maintainers (if any).

Exceptionally the core maintainers can propose another maintainer, core or module maintainer, to stepdown. All core maintainers must unanimously agree before making any pull request.

In both cases, we encourage the maintainers to give a brief explanation of their reasons to step down.

Making a release

1. **Cross-check CI green lights.** Are all Travis builds green? Is nightly Jenkins build green?
2. **Cross-check Read The Docs documentation builds.** Are docs building fine? Is this check done as part of the continuous integration? If not, add it.
3. **Cross-check demo site builds.** Is demo site working?
5. **Check author list.** Are all committers listed in AUTHORS file? Use `kwalitee check authors`. Add newcomers. Is this check done as part of the continuous integration? If not, add it.
6. **Update I18N message catalogs.**
7. **Update version number.** Stick to [semantic versioning](#).
8. **Generate release notes.** Use `kwalitee prepare release v1.1.0.` to generate release notes. Use empty commits with “AMENDS” to amend wrong past messages before releasing.
10. **Tag it. Push it.** Sign the tag with your GnuPG key. Push it to PyPI. Is the PyPI deployment done automatically as part of the continuous integration? If not, add it.
11. **Bump it.** Don’t forget the issue the pull request with a post-release version bump. Use `.devYYYYMMDD` suffix.

How to setup a new repository

Understanding branches

The official Invenio repository contains several branches for maintenance and development purposes. We roughly follow the usual git model as described in [man 7 gitworkflows](#) and elsewhere.

In summary, the new patchlevel releases (X.Y.Z) happen from the `maint` branch, the new minor feature releases (X.Y) happen from the `master` branch, and new major feature releases (X) happen after they mature in the optional `next` branch. A more detailed description follows.

`maint`

This is the maintenance branch for the latest stable release. There can be several maintenance branches for every release series (`maint-0.99`, `maint-1.0`, `maint-1.1`), but typically we use only `maint` for the latest stable release.

The code that goes to the maintenance branch is of bugfix nature only. It should not alter DB table schema, Invenio config file schema, local configurations in the `etc` folder or template function parameters in a backward-incompatible way. If it contains any new features, then they are switched off in order to be fully compatible with the previous releases in this series. Therefore, for installations using any Invenio released X.Y series, it should be always safe to upgrade the system at any moment in time by (1) backing up their `etc` folder containing local configuration, (2) installing the corresponding `maint-X.Y` branch updates, and (3) rolling back the `etc` folder with their customizations. This upgrade process will be automatized in the future via special `inveniomange` options.

`master`

The `master` branch is where the new features are being developed and where the new feature releases are being made from. The code in `master` is reviewed and verified, so that it should be possible to make a new release out of this branch almost at any given point in time. However, Invenio installations that would like to track this branch should be aware that DB table definitions are not frozen and may change, the config is not frozen and may change, etc, until the release time. So while `master` is relatively stable for usage, it should be treated with extreme care, because updates between day D1 and day D2 may require DB schema and `etc` configuration changes that are not covered by usual

`inveniomanage` update statements, so people should be prepared to study the differences and update DB schemata and config files themselves.

Integrating

Integrating principles

1. **Check it from the helicopter.** If it ain't green, it ain't finished. If it ain't understandable, it ain't documented.
2. **Beware of inter-module relations.** Changing API? Perhaps this pull request may break other modules. Check outside usage. Check the presence of `versionadded`, `versionmodified`, `deprecated` docstring directives.
3. **Beware of inter-service relations.** Changing pre-existing tests? Perhaps this pull request does not fit the needs of other Invenio services.
6. **Avoid self merges.** Each pull request should be seen by another pair of eyes. Was it authored and reviewed by two different persons? Good. Were the two different persons coming from two different service teams? Better.

Triaging

Triaging principles

The purpose of [issue triage process](#) is to make sure all the Invenio issues and tasks are well described, sorted out according to priorities into timely milestones, and that they have well assigned a responsible person to tackle them. The triage process is done collectively by the representatives of various Invenio distributed teams. The triage team members make sure of the big picture, cross-consider issues and tasks among interested services, help to note down, understand, prioritise, and follow-upon them.

1. **Every issue should have an assignee.** Issues without assignees are sad and likely to remain so. Each issue should have a driving force behind it.
2. **Every issue should have a milestone.** Issues without milestones are sad and likely to remain unaddressed.
3. **Use additional type and priority labels.** Helps to find related issues out quickly.
4. **Use additional service labels.** Helps to distinguish which issues are of utmost importance to which services. Helps to keep a service-oriented dashboard overview.
6. **Nobody in sight to work on this? Discuss and triage.** The triage team should take action if there is no natural candidate to work on an issue.
7. **No time to realistically tackle this? Use “Someday” or close.** We don't want to have thousands of open issues. Issues from the someday open or closed pool can be revived later, should realistic manpower be found.

Triaging team

Invenio triaging team consists of persons who manipulate issues, create labels, attribute assignments, follow up progress, and collaboratively decide on milestones for various Invenio projects on GitHub.

To enter a triaging team, the following conditions must be met:

1. The person is an established Invenio scrum master or developer who knows well the wider project ecosystem. E.g. worked for more than N1 years on the project OR maintained more than N2 modules OR committed more than N3 lines of code to N4 different modules. (For respectably large values of N.)

2. The person will use triaging rights for manipulating tickets only; never for pushing code. Since GitHub ticket-manipulation rights are indistinguishable from GitHub code-pushing rights, the persons in the “Triagers” team should take extra care to avoid accidental mishaps in this direction. (Such as doing `git push myorigin master -f` and realising only afterwards that “oops myorigin was wrongly set so now I have accidentally replaced the whole Invenio canonical project commit history”.) The code pushing is being done by the “Integrators”, not by the “Triagers”.
3. The person will endorse social contract related to participation in the collaborative consensual triaging and milestone decision-making process. The triaging team will represent various Invenio developer teams and services, participating together with the release management team on decisions related to the roadmap milestones and releases.

Triaging process

The process of triaging issues is being done continuously. This usually means that labels are being attached to issues, assignments are being decided, progress is being tracked, dependencies being clarified, questions being answered, etc.

The triaging team meets regularly (say once every 1-2 weeks) to go over the list of open issues for any catch-up, follow-up, and re-classification of issues with respect to milestones.

Issue labels

The issues are attributed namespaced labels indicating the following issue properties:

- type:
 - *t_bug*: bug fix
 - *t_enhancement*: new feature or improvement of existing feature
- priority:
 - *p_blocker*: highest priority, e.g. breaks home page
 - *p_critical*: higher priority, e.g. test case broken
 - *p_major*: normal priority, used for most tickets
 - *p_minor*: less priority, less visible user impact
 - *p_trivial*: lowest priority, cosmetics
- component:
 - *c_WebSearch*: search component
 - *c_WebSubmit*: submit component
 - etc
- status:
 - *in_work*: developer works on the topical branch
 - *in_review*: reviewer works on reviewing the work done
 - *in_integration*: integrator works on checking interplay with other services
- resolution:
 - *r_fixed*: issue fixed
 - *r_invalid*: issue is not valid

- *r_wontfix*: issue will not be dealt with for one reason or another
- *r_duplicate*: issue is a duplicate of another issue
- *r_question*: issue is actually a user question
- *r_rfc*: issue is actually an open RFC
- branch:
 - *maint-x.y*: issue applies to Invenio maint-x.y branch
 - *master*: issue applies to Invenio master branch
 - *next*: issue applies to Invenio next branch
 - *pu*: issue applies to Invenio pu branch
- version:
 - *v0.99.1*: issue was observed on version v0.99.1
 - *v1.1.3*: issue was observed on version v1.1.3
 - etc

The label types and values are coming from our Trac past and may be amended e.g. to take into account new component names in Invenio v2.0.

Milestones and releases

Issues may be attributed milestones that are closely related with feature-dependent and/or time-dependent release schedule of Invenio releases.

There are two kinds of milestones: “release-oriented” milestones (say v1.1.7) and “someday” milestones (say v1.1.x) for each given release series (v1.1 in this case). The release-oriented milestones may have dates attached to them; the someday milestone may not.

Typically, a new issue is given (a) the closest milestone in the given release series if its urgency is high, or (b) a later milestone in the given release series depending on the estimated amount of work and available resources, or is (c) left in the catch-all someday milestone out of which the issue can be later cherry-picked and moved to one of the concrete release-oriented milestones depending on available resources.

Example: after Invenio v1.4.0 is released, all incoming bug reports for this version will go to the “someday” milestone for this release series, i.e. to “v1.4.x”. A new XSS vulnerability issue will go straight to the next milestone “v1.4.1” because its release is urgent. A typo in an English output phrase in the basket module will remain in the someday milestone “v1.4.x” until it is picked for one of later releases, say v1.4.7, depending on available resources in the basket team.

The triaging team together with the release management team will periodically review issues in a given release series and decide upon the set of issues going into a concrete release-oriented milestone (say these 15 issues for v1.4.1 milestone) after which the issue set is frozen and a sprint may be co-organised to meet the target deadline. Once all the issues have been solved, a new Invenio bug-fix release v1.4.1 is published and the release-oriented triaging cycle starts anew with v1.4.2.

(Note that someday milestones are usually more useful for new feature releases; they might remain relatively empty for bug fix releases.)

Releases

We follow [semantic versioning](#) and [PEP-0440](#) release numbering practices.

Invenio v3.x

Unreleased

Invenio v3.0 will be released when the Invenio code base is fully split into a set of standalone independent Python packages.

Invenio v2.x

Semi-stable

Invenio v2.x code base is a hybrid architecture that uses Flask web development framework combined with Invenio v1.x framework.

Note: The 2.x code base is not suitable for production systems, and will not receive any further development nor security fixes.

Released versions include:

Invenio v2.1:

- v2.1.1 - released 2015-09-01
- v2.1.0 - released 2015-06-16

Invenio v2.0:

- v2.0.6 - released 2015-09-01
- v2.0.5 - released 2015-07-17
- v2.0.4 - released 2015-06-01
- v2.0.3 - released 2015-05-15
- v2.0.2 - released 2015-04-17
- v2.0.1 - released 2015-03-20
- v2.0.0 - released 2015-03-04

Invenio v1.x

Stable

Invenio v1.x code base is suitable for stable production. It uses legacy technology and custom web development framework.

Note: Invenio v1.x is in feature freeze and will only receive important bug and security fixes.

Released versions include:

Invenio v1.2:

- v1.2.2 - released 2016-11-25
- v1.2.1 - released 2015-05-21
- v1.2.0 - released 2015-03-03

Invenio v1.1:

- v1.1.7 - released 2016-11-20
- v1.1.6 - released 2015-05-21
- v1.1.5 - released 2015-03-02
- v1.1.4 - released 2014-08-31
- v1.1.3 - released 2014-02-25
- v1.1.2 - released 2013-08-19
- v1.1.1 - released 2012-12-21
- v1.1.0 - released 2012-10-21

Invenio v1.0:

- v1.0.10 - released 2016-11-09
- v1.0.9 - released 2015-05-21
- v1.0.8 - released 2015-03-02
- v1.0.7 - released 2014-08-31
- v1.0.6 - released 2014-01-31
- v1.0.5 - released 2013-08-19
- v1.0.4 - released 2012-12-21
- v1.0.3 - released 2012-12-19
- v1.0.2 - released 2012-10-19
- v1.0.1 - released 2012-06-28
- v1.0.0 - released 2012-02-29
- v1.0.0-rc0 - released 2010-12-21

Invenio v0.x

Invenio v0.x code base was developed and used in production instances since 2002. The code base is interesting only for archaeological purposes.

Released versions include:

- v0.99.9 - released 2014-01-31
- v0.99.8 - released 2013-08-19
- v0.99.7 - released 2012-12-18
- v0.99.6 - released 2012-10-18
- v0.99.5 - released 2012-02-21
- v0.99.4 - released 2011-12-19
- v0.99.3 - released 2010-12-13
- v0.99.2 - released 2010-10-20
- v0.99.1 - released 2008-07-10

- v0.99.0 - released 2008-03-27
- v0.92.1 - released 2007-02-20
- v0.92.0. - released 2006-12-22
- v0.90.1 - released 2006-07-23
- v0.90.0 - released 2006-06-30
- v0.7.1 - released 2005-05-04
- v0.7.0 - released 2005-04-06
- v0.5.0 - released 2004-12-17
- v0.3.3 - released 2004-07-16
- v0.3.2 - released 2004-05-12
- v0.3.1 - released 2004-03-12
- v0.3.0 - released 2004-03-05
- v0.1.2 - released 2003-12-21
- v0.1.1 - released 2003-12-19
- v0.1.0 - released 2003-12-04
- v0.0.9 - released 2002-08-01

Code of Conduct

We endorse the [Python Community Code of Conduct](#):

The Invenio community is made up of members from around the globe with a diverse set of skills, personalities, and experiences. It is through these differences that our community experiences great successes and continued growth. When you're working with members of the community, we encourage you to follow these guidelines which help steer our interactions and strive to keep Invenio a positive, successful, and growing community.

A member of the Invenio community is:

1. **Open.** Members of the community are open to collaboration, whether it's on RFCs, patches, problems, or otherwise. We're receptive to constructive comment and criticism, as the experiences and skill sets of other members contribute to the whole of our efforts. We're accepting of all who wish to take part in our activities, fostering an environment where anyone can participate and everyone can make a difference.
2. **Considerate.** Members of the community are considerate of their peers – other Invenio users. We're thoughtful when addressing the efforts of others, keeping in mind that often times the labor was completed simply for the good of the community. We're attentive in our communications, whether in person or online, and we're tactful when approaching differing views.
3. **Respectful.** Members of the community are respectful. We're respectful of others, their positions, their skills, their commitments, and their efforts. We're respectful of the volunteer efforts that permeate the Invenio community. We're respectful of the processes set forth in the community, and we work within them. When we disagree, we are courteous in raising our issues.

Overall, we're good to each other. We contribute to this community not because we have to, but because we want to. If we remember that, these guidelines will come naturally.

We recommend the “egoless” programming principles (Gerald Weinberg, [The Psychology of Computer Programming](#), 1971):

1. **Understand and accept that you will make mistakes.** The point is to find them early, before they make it into production. Fortunately, except for the few of us developing rocket guidance software at JPL, mistakes are rarely fatal in our industry, so we can, and should, learn, laugh, and move on.
2. **You are not your code.** Remember that the entire point of a review is to find problems, and problems will be found. Don’t take it personally when one is uncovered.
3. **No matter how much “karate” you know, someone else will always know more.** Such an individual can teach you some new moves if you ask. Seek and accept input from others, especially when you think it’s not needed.
4. **Don’t rewrite code without consultation.** There’s a fine line between “fixing code” and “rewriting code.” Know the difference, and pursue stylistic changes within the framework of a code review, not as a lone enforcer.
5. **Treat people who know less than you with respect, deference, and patience.** Nontechnical people who deal with developers on a regular basis almost universally hold the opinion that we are prima donnas at best and crybabies at worst. Don’t reinforce this stereotype with anger and impatience.
6. **The only constant in the world is change.** Be open to it and accept it with a smile. Look at each change to your requirements, platform, or tool as a new challenge, not as some serious inconvenience to be fought.
7. **The only true authority stems from knowledge, not from position.** Knowledge engenders authority, and authority engenders respect – so if you want respect in an egoless environment, cultivate knowledge.
8. **Fight for what you believe, but gracefully accept defeat.** Understand that sometimes your ideas will be overruled. Even if you do turn out to be right, don’t take revenge or say, “I told you so” more than a few times at most, and don’t make your dearly departed idea a martyr or rallying cry.
9. **Don’t be “the guy in the room”.** Don’t be the guy coding in the dark office emerging only to buy cola. The guy in the room is out of touch, out of sight, and out of control and has no place in an open, collaborative environment.
10. **Critique code instead of people** – be kind to the coder, not to the code. As much as possible, make all of your comments positive and oriented to improving the code. Relate comments to local standards, program specs, increased performance, etc.

License

Invenio is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

Invenio is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Invenio; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

In applying this license, CERN does not waive the privileges and immunities granted to it by virtue of its status as an Intergovernmental Organization or submit itself to any jurisdiction.

Authors

See *Communication channels* for how to get in touch with us.

Active and past contributors:

- Adrian Pawel Baran <adrian.pawel.baran@cern.ch>
- Adrian-Tudor Panescu <adrian.tudor.panescu@cern.ch>
- Alberto Pepe <alberto.pepe@cern.ch>
- Alessio Deiana <alessio.deiana@cern.ch>
- Alexander Wagner <alexander.wagner@desy.de>
- Alexandra Silva <xana@correio.ci.uminho.pt>
- Alper Cinar <alper@srdc.com.tr>
- Anna Afshar <anna.afsharghasemlouy@epfl.ch>
- Annette Holtkamp <annette.holtkamp@cern.ch>
- Antonios Manaras <antonios.manaras@cern.ch>
- Artem Tsikiridis <artem.tsikiridis@cern.ch>
- Avraam Tsantekidis <avraam.tsantekidis@cern.ch>
- Axel Voitier <axel.voitier@gmail.com>
- Benoit Thiell <benoit.thiell@cern.ch>
- Björn Oltmanns <bjoern.oltmanns@gmail.com>
- Carmen Alvarez Perez <carmen.alvarez.perez@cern.ch>
- Christopher Dickinson <christopher.dickinson@cern.ch>
- Christopher Hayward <christopher.james.hayward@cern.ch>
- Christopher Parker <chris.parker.za@gmail.com>
- Cornelia Plott <c.plott@fz-juelich.de>
- Cristian Bacchi <cristian.bacchi@gmail.com>
- Dan Michael O. Heggø <danmichaelo@gmail.com>
- Daniel Lovasko <daniel.lovasko@cern.ch>
- Daniel Stanculescu <daniel.stanculescu@cern.ch>
- David Bengoa <david.bengoa.rocandio@cern.ch>
- Diane Berkovits <diane.berkovits@cern.ch>
- Dimitrios Semitsoglou-Tsiapos <dsemitso@cern.ch>
- Dinos Kousidis <konstantinos.kousidis@cern.ch>
- Eamonn Maguire <eamonnmag@gmail.com>
- Eduardo Benavidez <eduardo.benavidez@gmail.com>
- Eduardo Margallo <eduardo.margallo@cern.ch>
- Eirini Psallida <eirini.psallida@cern.ch>
- Eric Stahl <eric.stahl@utbm.fr>
- Erik Simon <erik.simon@unine.ch>
- Esteban J. G. Gabancho <esteban.jose.garcia.gabancho@cern.ch>

- Evelthon Prodromou <epro@prodromou.eu>
- Fabio Souto <fsoutomoure@gmail.com>
- Federico Poli <federico.poli@cern.ch>
- Ferran Jorba <Ferran.Jorba@uab.cat>
- Flavio Costa <flavio.costa@cern.ch>
- Franck Grenier <franckgrenier@wanadoo.fr>
- Frederic Gobry <frederic.gobry@epfl.ch>
- Fredrik Nygård Carlsen <me@frecar.no>
- Gabriel Hase <gabriel.hase@cern.ch>
- Georgios Kokosioulis <giokokos@gmail.com>
- Georgios Papoutsakis <georgios.papoutsakis@cern.ch>
- Gerrit Rindermann <Gerrit.Rindermann@cern.ch>
- giannitsan <tsanaks.10@gmail.com>
- Gilles Louppe <g.louppe@gmail.com>
- Giovanni Di Milia <gdimilia@cfa.harvard.edu>
- Glenn Gard <glenn4@aol.com>
- Graham R. Armstrong <graham.richard.armstrong@cern.ch>
- Gregory Favre <gregory.favre@cern.ch>
- Grzegorz Szpura <grzegorz.szpura@cern.ch>
- Guillaume Lastecoueres <PX9e@gmx.fr>
- Guotie <guotie.9@gmail.com>
- Harris Tzovanakis <me@drjova.com>
- Hector Sanchez <hector.sanchez@cern.ch>
- Henning Weiler <henning.weiler@cern.ch>
- Ivan Masár <helix84@centrum.sk>
- Ivan Masár <helix84@centrum.sk>
- Jacopo Notarstefano <jacopo.notarstefano@cern.ch>
- Jaime Garcia Llopis <jaime.garcia.llopis@cern.ch>
- Jake Cowton <jake.calum.cowton@cern.ch>
- Jan Aage Lavik <jan.age.lavik@cern.ch>
- Jan Brice Krause <jan.brice.krause@cern.ch>
- Jan Iwaszkiewicz <jan.iwaszkiewicz@cern.ch>
- Jan Stypka <jan.stypka@cern.ch>
- Javier Martin <javier.martin.montull@cern.ch>
- Jay Luker <lbjay@reallywow.com>
- Jerome Caffaro <jerome.caffaro@cern.ch>

- Jiri Kuncar <jiri.kuncar@cern.ch>
- João Batista <jnfbatista@gmail.com>
- Joaquim Rodrigues Silvestre <joaquim.rodrigues.silvestre@cern.ch>
- Jocelyne Jerdelet <jocelyne.jerdelet@cern.ch>
- Jochen Klein <klein.jochen@gmail.com>
- Joe Blaylock <jrbl@slac.stanford.edu>
- Joe MacMahon <joe.macmahon@cern.ch>
- Joël Vogt <joel.vogt@unifr.ch>
- Johann C. Rocholl <johann@browsershots.org>
- Johnny Mariéthoz <johnny.mariethoz@rero.ch>
- Jorge Aranda Sumarroca <jorge.aranda.sumarroca@cern.ch>
- Juan Francisco Pereira Corral <juan.francisco.pereira.corral@cern.ch>
- Julio Pernia Aznar <jpernia@altransdb.com>
- Juliusz Sompolski <julsomp@gmail.com>
- Jurga Girdzijauskaitė <jurga.gird@gmail.com>
- Kamil Neczaj <kamil.neczaj@cern.ch>
- Kenneth Hole <kenneth@tind.io>
- Kevin Bowrin <kjbowrin@gmail.com>
- Kevin M. Flannery <flannery@fnal.gov>
- Kevin Sanders <kevin.sanders@cern.ch>
- Kirsten Sachs <kirsten.sachs@desy.de>
- Konstantinos Kostis <konstantinos.kostis@cern.ch>
- Konstantinos Kousidis <dinosimpson@pb-d-128-141-29-229.cern.ch>
- Konstantinos Ntemagkos <konstantinos.ntemagkos@cern.ch>
- Krzysztof Jedrzejek <krzysztof.jedrzejek@cern.ch>
- Krzysztof Lis <krzysztof.lis@cern.ch>
- Kyriakos Liakopoulos <kyriakos.liakopoulos@cern.ch>
- Lars Christian Raae <lars.christian.raae@cern.ch>
- Lars Holm Nielsen <lars.holm.nielsen@cern.ch>
- Laura Rueda <laura.rueda@cern.ch>
- Leonardo Rossi <leonardo.r@cern.ch>
- Lewis Barnes <lewis.barnes@cern.ch>
- Ludmila Marian <ludmila.marian@gmail.com>
- Luke Andrew Smith <smithey_72@hotmail.com>
- Maja Gracco <maja.gracco@cern.ch>
- Marco Neumann <marco@crepererum.net>

- Marcus Johansson <marcus.johansson@cern.ch>
- Marios Kogias <marioskogias@gmail.com>
- Marko Niinimäki <manzikki@gmail.com>
- Markus Goetz <murxman@gmail.com>
- Martin Vesely <martin.vesely@cern.ch>
- Mateusz Susik <mateusz.susik@cern.ch>
- Mathieu Barras <mbarras@gmail.com>
- Miguel Martín <miguelm@unizar.es>
- Miguel Martinez Pedreira <miguel.martinez.pedreira@cern.ch>
- Mikael Karlsson <i8myshoes@gmail.com>
- Mikael Vik <mikael.vik@cern.ch>
- Mike Marino <mmarino@gmail.com>
- Mike Sullivan <sul@slac.stanford.edu>
- Minn Soe <minn.soe@cern.ch>
- Nicholas Robinson <nicholas.robinson@cern.ch>
- Nicolas Harraudeau <nicolas.harraudeau@cern.ch>
- Nikola Yolov <nikola.yolov@cern.ch>
- Nikolaos Kalodimas <nikolaos.kalodimas@cern.ch>
- Nikolaos Kasioumis <nikolaos.kasioumis@cern.ch>
- Nikolay Dyankov <ndyankov@gmail.com>
- Nino Jejelava <nino.jejelava@gmail.com>
- Olivier Canévet <olivier.canevet@cern.ch>
- Olivier Serres <olivier.serres@gmail.com>
- Øystein Blixhavn <oystein@blixhavn.no>
- Øyvind Østlund <oyvind.ostlund@cern.ch>
- Pablo Vázquez Caderno <pcaderno@cern.ch>
- Pamfilos Fokianos <pamfilos.fokianos@cern.ch>
- Patrick Glauner <patrick.oliver.glauner@cern.ch>
- Paulo Cabral <paulo.cabral@cern.ch>
- Pedro Gaudencio <pmgaudencio@gmail.com>
- Peter Halliday <phalliday@cornell.edu>
- Petr Brož <petr.broz@heaven-industries.com>
- Petros Ioannidis <petros.ioannidis@cern.ch>
- Piotr Praczyk <piotr.praczyk@piotr.praczyk@gmail.com>
- Radoslav Ivanov <radoslav.ivanov@cern.ch>
- Raja Sripada <raja.sripada@cern.ch>

- Raquel Jimenez Encinar <raquel.jimenez.encinar@cern.ch>
- Richard Owen <ro@tes.la>
- Roberta Faggian <roberta.faggian@cern.ch>
- Roman Chyla <roman.chyla@cern.ch>
- Ruben Pollan <ruben.pollan@cern.ch>
- Sami Hiltunen <sami.mikael.hiltunen@cern.ch>
- Samuele Carli <samuele.carli@cern.ch>
- Samuele Kaplun <samuele.kaplun@cern.ch>
- Sebastian Witowski <sebastian.witowski@cern.ch>
- Stamen Todorov Peev <stamen.peev@cern.ch>
- Stefan Hesselbach <s.hesselbach@gsi.de>
- Stephane Martin <stephane.martin@epfl.ch>
- Theodoropoulos Theodoros <theod@lib.auth.gr>
- Thierry Thomas <thierry@FreeBSD.org>
- Thomas Baron <thomas.baron@cern.ch>
- Thomas Karampelas <thomas.karampelas@cern.ch>
- Thomas McCauley <thomas.mccauley@cern.ch>
- Thorsten Schwander <thorsten.schwander@gmail.com>
- Tiberiu Dondera <tiberiu.dondera@pronet-consulting.com>
- Tibor Simko <tibor.simko@cern.ch>
- Tony Ohls <tony.ohls@cern.ch>
- Tony Osborne <tony.osborne@cern.ch>
- Travis Brooks <travis@slac.stanford.edu>
- Trond Aksel Myklebust <trond.aksel.myklebust@cern.ch>
- Valkyrie Savage <vasavage@gmail.com>
- Vasanth Venkatraman <vasanth.venkatraman@cern.ch>
- Vasyl Ostrovskyi <vo@imath.kiev.ua>
- Victor Engmark <victor.engmark@cern.ch>
- Wojciech Ziolk <wojciech.ziolk@cern.ch>
- Yannick Tapparel <yannick.tapparel@cern.ch>
- Yoan Blanc <yoan.blanc@cern.ch>
- Yohann Paris <yohann.paris@cern.ch>
- Željko Kraljević <w.kraljevic@gmail.com>

See also THANKS file.

TODO

Todo

Make better description of project website. Add high level road map etc to project website.

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/invenio/checkouts/iugw2017/docs/community/channel.rst`, line 55.)

Todo

Setup an architecture team meeting between core maintainers.

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/invenio/checkouts/iugw2017/docs/community/channel.rst`, line 102.)

Todo

Fill with new proposal

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/invenio/checkouts/iugw2017/docs/community/maintainers_guide/index.rst`, line 4.)

Todo

Needs to be combined from existing parts and write the missing parts.

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/invenio/checkouts/iugw2017/docs/community/style_guide.rst`, line 16.)

Todo

Write this section

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/invenio/checkouts/iugw2017/docs/introduction/digital_invenio.rst`, line 21.)

Todo

Fix up this section

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/invenio/checkouts/iugw2017/docs/introduction/index.rst`, line 23.)

Todo

Write this section

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/invenio/checkouts/iugw2017/docs/introduction/index.rst line 107.)

Todo

Write this section

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/invenio/checkouts/iugw2017/docs/introduction/usage.rst line 4.)

Todo

Describe records, files, buckets.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/invenio/checkouts/iugw2017/docs/usersguide/loading.rst line 121.)

I

INVENIO_ELASTICSEARCH_HOST, **9, 51**
INVENIO_POSTGRESQL_DBNAME, **9, 51**
INVENIO_POSTGRESQL_DBPASS, **9, 51**
INVENIO_POSTGRESQL_DBUSER, **9, 51**
INVENIO_POSTGRESQL_HOST, **9, 51**
INVENIO_RABBITMQ_HOST, **9, 51**
INVENIO_REDIS_HOST, **9, 51**
INVENIO_USER_EMAIL, **9, 51**
INVENIO_USER_PASS, **9, 51**
INVENIO_WEB_HOST, **9, 51**
INVENIO_WEB_INSTANCE, **9, 51**
INVENIO_WEB_VENV, **9, 51**
INVENIO_WORKER_HOST, **9, 51**